VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DISTRIBUOVANÁ OBNOVA HESEL S VYUŽITÍM NÁSTROJE HASHCAT

DISTRIBUTED PASSWORD RECOVERY USING HASHCAT TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. LUKÁŠ ZOBAL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. RADEK HRANICKÝ**
SUPERVISOR

BRNO 2018

Zadání diplomové práce/21072/2017/xzobal00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Zobal Lukáš, Bc.**
Obor: Bezpečnost informačních technologií
Téma: **Distribuovaná obnova hesel s využitím nástroje hashcat**
Distributed Password Recovery Using Hashcat Tool
Kategorie: Paralelní a distribuované výpočty

Pokyny:

1. Seznamte se s nástrojem hashcat pro obnovu hesel a nástrojem BOINC užívaným pro "grid computing"
2. Prostudujte podporované útoky v nástroji hashcat a možnosti jejich využití v distribuovaném prostředí
3. Navrhněte úpravy současného software Fitcrack, které zajistí funkčnost s nástrojem hashcat, se zaměřením na serverovou část.
4. Návrh implementujte a experimentálně ověřte jeho funkčnost.
5. Zhodnoťte dosažené výsledky a navrhněte možná budoucí rozšíření.

Literatura:

- D. P. Anderson, "BOINC: a system for public-resource computing and storage," Fifth IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4-10. doi: 10.1109/GRID.2004.14.
- D. P. Anderson, E. Korpela and R. Walton, "High-performance task distribution for volunteer computing," First International Conference on e-Science and Grid Computing (e-Science'05), Melbourne, Vic., 2005, pp. 8 pp.-203. doi: 10.1109/E-SCIENCE.2005.51.
- HRANICKÝ Radek, HOLKOVIČ Martin, MATOUŠEK Petr a RYŠAVÝ Ondřej. On Efficiency of Distributed Password Recovery. The Journal of Digital Forensics, Security and Law. 2016, roč. 11, č. 2, s. 79-96. ISSN 1558-7215.
- HRANICKÝ Radek, ZOBAL Lukáš, VEČEŘA Vojtěch a MATOUŠEK Petr. Distributed Password Cracking in a Hybrid Environment. In: Proceedings of SPI 2017. Brno: Universita Obrany v Brně, 2017, s. 75-90. ISBN 978-80-7231-414-0.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hranický Radek, Ing.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem této práce je vyvinout distribuované řešení pro obnovu hesel využívající nástroje hashcat. Základem tohoto řešení je nástroj pro obnovu hesel, Fitcrack, vyvinutý v rámci mé předchozí spolupráce na projektu TARZAN. Distribuce práce mezi jednotlivé výpočetní uzly bude řešena pomocí systému BOINC, který je hojně využíván pro dobrovolnické poskytování výpočetní síly pro různé vědecké projekty. Výsledkem je pak nástroj, který používá robustní a spolehlivý systém distribuce práce klientům napříč lokální sítí nebo sítí Internet. Na nich probíhá obnova přidělených hesel a kryptografických hešů rychlým a efektivním způsobem, s využitím standardu OpenCL pro akceleraci celého procesu na principu GPGPU.

Abstract

The aim of this thesis is a distributed solution for password recovery, using hashcat tool. The basis of this solution is password recovery tool Fitcrack, developed during my previous work on TARZAN project. The jobs distribution is done using BOINC platform, which is widely used for volunteer computing in a variety of scientific projects. The outcome of this work is a tool, which uses robust and reliable way of job distribution across a local or the Internet network. On the client side, fast and efficient password recovery process takes place, using OpenCL standard for acceleration of the whole process with the use of GPGPU principle.

Klíčová slova

obnova hesel, lámání hesel, bezpečnost, hashcat, Fitcrack, GPGPU, OpenCL, BOINC, distribuované prostředí

Keywords

password recovery, password cracking, security, hashcat, Fitcrack, GPGPU, OpenCL, BOINC, distributed environment

Citace

ZOBAL, Lukáš. *Distribuovaná obnova hesel s využitím nástroje hashcat.* Brno, 2018. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Hranický

Distribuovaná obnova hesel s využitím nástroje hashcat

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Radka Hranického. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Zobal
20. května 2018

Poděkování

Tímto bych chtěl poděkovat Ing. Radku Hranickému za vedení a odbornou pomoc při vypracování této diplomové práce. Dále také všem členům výzkumného projektu TARZAN, kteří mi v průběhu roku poskytovali cenné rady a inspiraci.

Obsah

Kapitola 1

Úvod

Hesla jsou dnes našimi každodenními společníky. I přes všechny své nevýhody jsou stále nejčastějším způsobem autentizace a pomocí nich přesvědčujeme své elektronické okolí o své identitě. Od zapnutí počítače, přihlašování se do sociálních sítí, emailu nebo internetového bankovnictví po přebírání balíku na poště prokazujeme naši identitu znalostí hesla. To však může být jednoduše prozrazeno, či dokonce uhodnuto, a způsobit tak nemalé potíže. Toho si odborníci všímají již desetiletí [14] a snaží se hesla nahradit chytřejšími způsoby autentizace, jakým je kupříkladu biometrie. To se začíná dařit hlavně v poslední době[1]. Spolehlivost takových systémů ale stále nedosahuje uspokojivých hodnot a pokud ano, pojí se s takovými metodami nespočet dalších problémů, jakou jsou vysoká cena, prozrazování soukromých informací jedince či nemožnost provozovat tuto metodu v určitém prostředí. Kvůli těmto a mnoha dalším důvodům [4] se hesla stále aktivně používají a v blízké budoucnosti nejspíše také budou [6].

Otázkou tak zůstává, jak zvolit bezpečné heslo, které nepůjde jednoduše uhodnout a zároveň bude člověk schopný zapamatovat si velké množství těchto hesel. Když si takové heslo tvoříme, systém nás většinou nutí dodržet jisté bezpečnostní prvky – heslo musí obsahovat velká i malá písmena, číslice či speciální znak a musí být určité délky. Na samotné osobě je pak vytvořit heslo tak, aby si jej zapamatovala. Tato kombinace končí ale často katastrofou. Klasický případ je volba slova ze slovníku nebo oblíbeného jména, nahrazení prvního písmene jeho velkou variantou, dále změna znaku „o“ za číslici „0“ a zakončení takto vytvořeného hesla znakem tečky. Systém vás v takovém případě pochválí za vytvoření velmi silného hesla. Opak je ale pravdou. Útočníci – nazývejme tak jedince, kteří si chtějí kvůli různým účelům přivlastnit cizí hesla – mají v dnešní době různé možnosti, jak vyzkoušet celou řadu postupů, kterými jejich oběť heslo vytvořila.

Ani doopravdy silná hesla však nemusí přinášet jistotu. Všechno totiž záleží na algoritmu vytvoření šifrovacího klíče. Výkon počítačů se navíc každým rokem rapidně zvyšuje. Algoritmy, které byly před deseti či dvaceti lety považovány za bezpečné, jsou dnes již nepoužitelné – ať už z hlediska krátkého šifrovacího klíče nebo objevení bezpečnostního nedostatku v těchto algoritmech. Dnes můžeme k prolomení hesla navíc využít nikoliv jediného počítače, ale distribuovaného systému, který obsahuje desítky, stovky či více strojů. Ty všechny pak běží na plný výkon za jediným cílem a tím je nalezení hledaného řetězce o několika málo znacích. V této práci využijeme systému BOINC, který je blíže popsaný v kapitole 2. Tento software nám dovolí distribuovat práci na předem neomezený počet stanic, které navíc mohou být rozmístěny po celém světě.

[1] https://w3c.github.io/webauthn/

Na samotných stanicích pak bude běžet nástroj hashcat, kterému se blíže věnujeme v kapitole 3. Jedná se o jeden z nejpokročilejších nástrojů v oblasti obnovy hesel, který kromě bezkonkurenční rychlosti poskytuje také řadu inteligentních způsobů generování hesel a možnost využití komplexních pravidel. Obnova probíhá s využitím standardu OpenCL, který umožňuje ověřování vygenerovaných kandidátů paralelizovat i na nejlepších grafických kartách dnešní doby a celý proces tak mnohonásobně urychlit.

Aktuální nástroj Fitcrack, jehož funkčnost stručně popíšeme v kapitole 4, má podobné schopnosti. Primární využití nástroje Fitcrack je v oblasti forenzní analýzy digitálních dat. Jeho hlavním nedostatkem oproti nástroji hashcat je však nedostatečná rychlost a nesrovnatelně méně podporovaných formátů. Naopak velkou výhodou je částečně funkční rozhraní pro komunikaci se zmíněným systémem BOINC, díky čemuž může být práce distribuována. Výkon nástroje Fitcrack se tak téměř lineárně zvyšuje s rostoucím počtem připojených klientů.

Cílem této práce je využít nástroje Fitcrack a nahradit klientskou část nástrojem hashcat. Výsledkem by pak byl robustní systém pro distribuci práce neomezenému množství klientů bez ohledu na jejich výkon a geografickou polohu, na nichž by probíhalo ověřování hesel nejrychlejším dostupným způsobem. Návrhem tohoto řešení se budeme zabývat v kapitole 5. Implementační detaily pak budou ukázány v kapitole 6.

V kapitole 7 experimentálně ověříme funkčnost jednotlivých distribuovaných útoků s využitím nástroje hashcat. Ukážeme také, jakého zrychlení jsme dosáhli právě díky využití nástroje hashcat na klientské straně našeho systému. Kapitola 8 pak uzavírá tuto práci. Zhodnotíme vytvořený nástroj a jeho schopnosti. Podíváme se také na možné nedostatky, potenciální rozšíření nástroje a další vývoj v budoucnu.

Kapitola 2

Systém BOINC

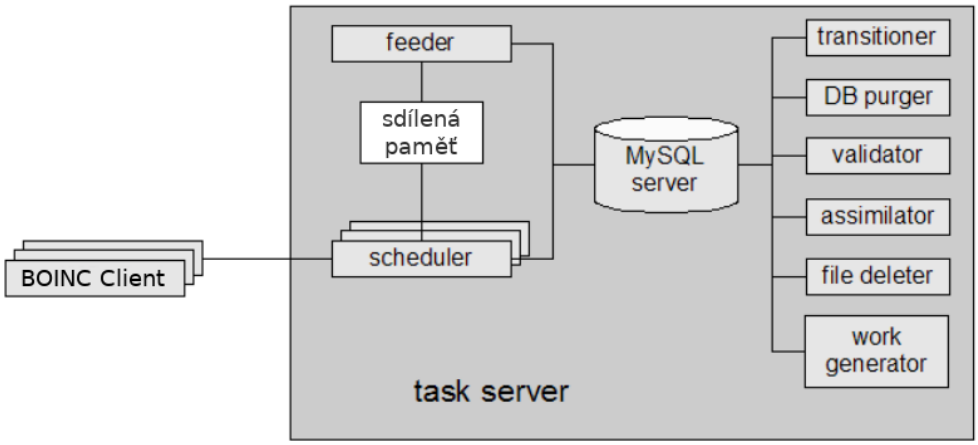
Dobrovolnické výpočty, anglicky *volunteer computing*, je princip výpočtu, kdy se na řešení určitého problému podílí dobrovolníci z geograficky oddělených lokací, kteří k tomu využívají své vlastní výpočetní prostředky. Potenciál tohoto přístupu je velmi vysoký a v budoucnu se bude pravděpodobně dále zvyšovat. Nacházíme se totiž v období, kdy největší výpočetní výkon není centralizovaný v superpočítačích, ale je dán stovkami milionů soukromých počítačů, konzolí a chytrých zařízení, jejichž výkon i počet stále rapidně roste. S jejich využitím je možné řešit do této doby nemožné problémy a navíc s velmi nízkým rozpočtem [3]. Zatímco stavba soukromých superpočítačů může být velmi nákladná, u dobrovolnického počítání záleží spíše na zájmu veřejnosti.

Systém BOINC vznikl právě s touto myšlenkou na Kalifornské univerzitě v Berkeley a to po zkušenostech z projektu SETI@home. Jednalo o zpracování velkého množství dat z radioteleskopů ve snaze získat důkazy o mimozemském životu [2]. Zde byla ve velkém měřítku nasazena technologie pro dobrovolnický výpočet a zájem lidí z celého světa byl obrovský. Postupem času začalo být navržené řešení nedostatečné a docházelo k častým problémům s přetížením nebo z hlediska podvádění, zasílání špatných výsledků apod.

BOINC umožňuje tvorbu samostatných projektů, které jsou zcela autonomní [1]. V rámci projektů je možné distribuovat několik různých aplikací a to na všechny možné platformy, včetně těch mobilních. Účastníci mohou být za účast na výpočtu také odměňováni a to speciálními kredity, které jsou rozdělovány na základě odvedené práce. To umožňuje následně sledovat statistiky, žebříčky a obecně to motivuje uživatele k pokračujícím výpočtům. Protože právě motivace uživatelů zapojit se do projektů rozhoduje o úspěchu či neúspěchu tohoto paradigmatu, poskytuje BOINC další řadu funkcí, jako je možnost sdružovat se do týmů, diskuse na fórech, soukromé profily, přívětivé uživatelské rozhraní, interaktivní spořič obrazovky či vlastní kryptoměna GridCoin[1], která se těží právě účastí na výpočtu pomocí nástroje BOINC. V neposlední řadě je také řešena otázka nedůvěryhodného prostředí – každý uživatel se totiž může kdykoliv odpojit nebo zaslat chybný výsledek, ať už z důvodu závady nebo s cílem zkazit výpočet.

Kromě dobrovolnického výpočtu je možné systém BOINC využít pro tzv. *grid computing*. I když cíl je společný, jsou mezi těmito dvěma přístupy určité rozdíly. *Grid computing* je situace, kdy práce není distribuována neznámým klientům, ale strojům, které jsou vlastněny jedincem, organizací či univerzitou a za které daný subjekt zodpovídá. Tím nám odpadá problém s nedůvěryhodným prostředím nebo motivací uživatelů. Nasazení klientů je většinou automatické a není třeba složité grafické rozhraní nebo spořiče obrazovky.

[1] Více informací o kryptoměně zde: http://www.gridcoin.us



Obrázek 2.1: Schéma plánovací části systému BOINC [1]

Architektura systému

Systém BOINC funguje na principu klient–server, kde každý klient periodicky komunikuje se serverem. Ten mu poskytuje pracovní jednotky (tzv. *workunits*) a přijímá od něj výsledky (tzv. *results*). Komunikace probíhá pomocí protokolu HTTP (volitelně také HTTPS) a zasílaných souborů ve formátu XML. Všechna data se následně soustřeďují v databázi MySQL. Funkce serveru systému BOINC je pak implementována sérií samostatných programů:

- **Work Generator** – Jedná se o démona na straně serveru, který běží v nekonečné smyčce a generuje úlohy pro jednotlivé klienty. Tento program je aplikačně závislý, administrátor každého projektu si ho tedy musí implementovat sám.
- **Assimilator** – Jedná se o démona na straně serveru, který má za úkol zpracovávat validní výsledky od klientů a provádět s nimi činnost závisející na administrátorech projektu. Může tedy například výsledky ukládat do separátní databáze, zobrazovat je správcům nebo je použít pro další výpočet.
- **Validator** – Toto je další BOINC démon, který má na starosti validaci příchozích výsledků. Kromě jejich syntaxe může ověřovat, zda se shodují s ostatními replikami dané úlohy a obsahují validní obsah.
- **File deleter** – Tato součást serveru maže vstupní a výstupní soubory každé úlohy po jejím dokončení.
- **Transitioner** – Jedná se o program, který udržuje dohled nad úlohami v databázi a reaguje na jejich modifikace příslušnou akcí.
- **Scheduler** – Tento program zajišťuje komunikaci s klienty. Přijímá jejich požadavky s výsledky a odesílá odpovědi se zadáním nových úloh, pokud byly nějaké vygenerovány.

- **Feeder** – Program Feeder úzce spolupracuje s programem Scheduler – poskytuje mu přístup do sdílené paměti a zajišťuje výlučný přístup ke sdíleným zdrojům.
- **Database purger** – Jedná se o pomocný program, který dokáže z databáze odstranit již dokončené úlohy, archivovat je ve formátu XML a volitelně také komprimovat.

Pro správce projektu jsou klíčové moduly Generator, Assimilator a Validator, které jsou aplikačně závislé. To znamená, že jejich funkčnost se liší v každém projektu a musí být tedy implementovány samotnými správci. Ostatní komponenty fungují vždy stejně a je tedy možné využít standardní implementaci poskytovanou systémem BOINC. Schéma systému BOINC se všemi podstatnými komponentami je vidět na obrázku 2.1.

Na klientské straně pak stačí nainstalovat nástroj BOINC Client a volitelně také grafickou nadstavbu BOINC Manager a připojit se k vybraným projektům. Pokud tyto projekty podporují platformu klienta, jsou automaticky zaslány všechna data a všechny programy nutné pro výpočet úlohy. Tyto úlohy mají většinou pevně definované datum a čas, do kdy musí být klientem zpracovány a výsledek odeslán. Pokud tomu tak není, je úloha přidělena jinému klientu. Při obdržení výsledku od klienta je výsledek validován a v pozitivním případě přidělen kredit.

Kapitola 3

Nástroj hashcat

Nástroj hashcat je nejrozšířenější nástroj pro obnovu hesel, který patří k těm nejrychlejším a nejpokročilejším v této oblasti. Podporuje více než 200 různých formátů a je schopen běžet na jakémkoliv zařízení podporujícím OpenCL, od CPU přes GPU až po FPGA. Je možné jej provozovat pod operačními systémy Windows, Linux i OSX. Kromě toho v poslední době přešel pod licenci MIT – je tedy open-source a jeho zdrojové kódy jsou veřejně přístupné[1]. Díky tomu se kolem nástroje hashcat sešla komunita expertů v dané oblasti, která posunuje vývoj stále vpřed a vytváří svá vlastní řešení, postavená právě na nástroji hashcat.

Mezi základní schopnosti tohoto nástroje patří vestavěný benchmark, který poskytuje údaj o přibližné rychlosti obnovy na daném stroji, dále možnost pozastavení výpočtu a vytvoření bodu obnovy nebo automatická kontrola teploty využívaného zařízení a případná korekce zátěže. Mezi elitu v oblasti obnovy hesel ho však řadí rychlost, kterou je schopen provádět ověřování jednotlivých hesel a to díky optimalizovaným OpenCL kernelům, pomocí kterých je možné celý proces paralelizovat na nejvýkonnějších grafických kartách dnešní doby. Dále také implementuje moderní přístupy ke generování hesel a různé druhy útoků, které si popíšeme v následujících kapitolách.

Přesto však nástroj hashcat není dokonalý a je stále ve vývoji. Základní testy s nástrojem ukázaly, že rychlosti naměřené benchmarkem jsou u několika formátů dost nepřesné. Tuto rychlost je navíc potřeba dále korigovat u formátů, kde je tato hodnota ovlivněna dalšími faktory, jako je například délka ověřovaného hesla u archivů 7-Zip. Při plánování úloh jednotlivým klientům budeme muset pracovat se stavovým prostorem zadaného útoku. Zde je další velká nevýhoda nástroje hashcat, který při práci neuvádí reálný stavový prostor, ale svůj vlastní, který se liší v závislosti na implementaci jednotlivých formátů. Značnou nevýhodou je také velmi omezené použití znakových sad obsahujících znaky mimo tabulku ASCII. S těmito a mnoha dalšími záludnosti se bude nutné vypořádat při následném návrhu distribuovaného řešení s využitím nástroje hashcat.

3.1 Základní typy útoků

Aktuální verze nástroje hashcat podporuje několik základních způsobů, kterými je možné generovat a následně ověřovat hesla. Tyto útoky se časem vyvíjely, některé z nich kvůli své neefektivnosti zanikly nebo se začlenily do jiných typů útoků, další se pak dají realizovat pomocí dodatečných nástrojů, z nichž některé si popíšeme níže v sekci 3.4. Tyto však většinou spočívají ve vygenerování slovníku hesel jistým speciálním způsobem. Takto

[1] Aktuální zdrojový kód nástroje hashcat na GitHubu: https://github.com/hashcat

vygenerované slovníky však musíme následně ověřit jedním ze základních útoků. Proto jsou tyto základní útoky pro naše účely esenciální – právě tyto způsoby běhu nástroje hashcat musíme detailně pochopit a následně jejich funkčnost implementovat v distribuovaném prostředí nástroje Fitcrack.

3.1.1 Slovníkový útok

Jeden ze základních útoků, který podporoval i samostatný nástroj Fitcrack, je slovníkový útok. Jedná se o situaci, kdy máme v textovém souboru uložená hesla, většinou každé na samostatném řádku, která chceme ověřit. Může se jednat například o nejčastěji využívaná hesla. Takových je možné najít velké množství a to z úniků různých internetových služeb. Nejznámějším takovým slovníkem je *rockyou.txt*, který vznikl na konci roku 2009 při krádeži osobních údajů ze stejnojmenné sítě[2]. Tento slovník obsahuje přibližně 15 milionů unikátních hesel. Pokud hledáme hesla vytvořená lidmi určité národnosti, může být vhodné použití slovníku hesel v daném jazyce. Pro Českou republiku můžeme jako příklad uvést únik emailů a hesel statisíců zákazníků internetového obchodu Mall.cz ze srpna 2017[3].

Kromě reálných dat je možné k získání relevantního slovníku hesel využít několik dalších praktik. Jedna z nich je analyzovat existující slovník hesel a na základě pravděpodobností jednotlivých znaků generovat taková hesla, která se velmi podobají těm reálným. Tato technika využívá tzv. Markovových řetězců a je již implementovaná v nástroji hashcat – více se jí budeme zabývat v sekci 3.2. Dalším způsobem je z existujícího slovníku vytvořit fragmenty hesel, skládající se například z malých či velkých písmen abecedy, čísel nebo speciálních znaků. Tyto fragmenty pak můžeme skládat libovolně za sebe a vytvářet tak další hesla. Experimentálně bylo ověřeno, že i takovýto přístup může být efektivní při hledání správného hesla [13]. Dalšími možnostmi pro generování slovníku hesel jsou například neuronové sítě a tzv. *deep learning*, který se ukazuje být srovnatelný s výše zmíněnými metodami a při kombinaci s Markovovými řetězci dokonce vykazuje mnohem lepší výsledky [7].

Samotný nástroj hashcat poskytuje několik dalších možností, jak hesla generovat. Tyto nástroje budou blíže popsány v sekci 3.4.

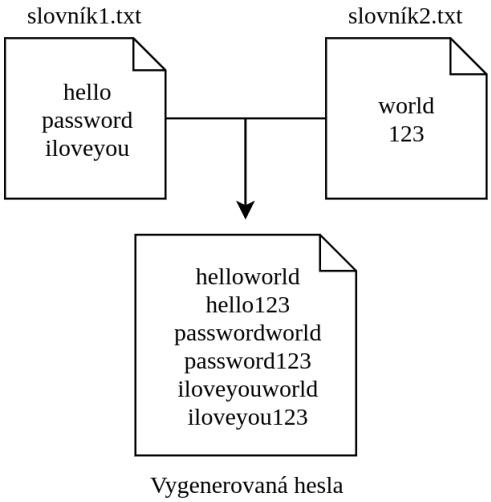
3.1.2 Kombinační útok

Princip tohoto útoku je velmi jednoduchý. Při spuštění kombinačního útoku poskytneme nástroji hashcat dva slovníky. Zkoušená hesla pak budou tvořena konkatenací hesel z prvního a druhého slovníku. Způsob funkčnosti tohoto útoku je ukázán na jednoduchém příkladu v diagramu 3.1.

Pro získání a použití vhodných slovníků zde platí stejné praktiky, které byly popsány v předchozí sekci o slovníkových útocích. Při použití tohoto typu útoku se nám však složitost zvýší tolikrát, kolik bude hesel ve druhém slovníku – obecně tedy z lineární na kvadratickou, a to zejména složitost časová. Složitost prostorová, tedy počet hesel, který musíme distribuovat, se při inteligentním plánování, které bude blíže popsáno při návrhu kombinačního útoku v kapitole 5.6, na kvadratickou složitost nezvýší.

[2] Více informací o úniku seznamu hesel v čitelné podobě uživatelů služby RockYou například zde: https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/

[3] Více informací o úniku hesel a emailů z českého internetového obchodu Mall.cz například zde: https://www.lupa.cz/aktuality/mall-cz-resetuje-hesla-k-casti-databaze-se-mohli-dostat-hackeri/



Obrázek 3.1: Ukázka funkčnosti kombinačního útoku

3.1.3 Maskový útok

Jedná se o základní útok, který v nástroji hashcat nahrazuje známý *brute-force*, tedy útok silou. Zatímco při útoku silou předpokládáme danou množinu znaků, danou délku hesla a našim cílem je ověřit všechny možnosti, maskový útok nám umožňuje omezit takovýto útok na jednotlivé znaky. Máme totiž možnost kontrolovat, které znaky se budou zkoušet na jednotlivých pozicích hesla a to zadáním tzv. masky. Důležité je, že pomocí masky můžeme útok silou napodobit, ale v naprosté většině případů existuje efektivnější způsob generování hesel, než zkoušet všechny možné hodnoty znaku.

Ukažme si to na příkladu, kde ono hledané heslo je *Jana94*. Při útoku silou bychom museli využít znakovou sadu obsahující jak malá i velká písmena latinky, tak čísla, celkově tedy $26 + 26 + 10 = 62$ znaků. Heslo je 6 znaků dlouhé. Předpokládejme, že hashcat dokáže vyzkoušet deset tisíc hesel za sekundu. Výpočet maximální doby obnovy takového hesla by potom vypadal následovně.

$$\frac{62^6}{10000s^{-1}} = 5680023s$$

Nalezení hesla v takovémto stavovém prostoru by tedy mohlo trvat přibližně až 66 dní, v závislosti na pořadí generovaných hesel. Nyní provedeme maskový útok, který vyžaduje určitou znalost o hledaném hesle. Určeme tedy první písmeno jako velké a poslední dva znaky jako číslici. To může být celkem častý způsob vytváření hesel lidskou bytostí, protože je takový tvar intuitivní a lehce zapamatovatelný. Výpočet maximální délky obnovy maskového útoku je následný.

$$\frac{26 \cdot 26^3 \cdot 10^2}{10000s^{-1}} = 4569s$$

V tomto případě bude celý stavový prostor hesel prohledán za 76 minut. Rozdíl je tedy obrovský. Tento příklad je zřejmě velmi specifický a vyžaduje znalost struktury hledaného hesla. Díky maskovému útoku však můžeme využít různých masek, které nám umožní vyzkoušet často využívané vzory v heslech, jako je výše uvedený příklad. U složitějších hesel je pak rozdíl ještě řádově větší.

Masku pro vytvoření výše zmíněného útoku je možné vidět na obrázku 3.2. Všimněme si, že i když ze začátku mohou generovaná hesla vypadat nesmyslně, postupně přijde i na česká slova či jména, na která můžeme právě tímto útokem mířit, tedy například spojení jména s rokem narození. Na takovýto případ je ještě vhodnější hybridní útok, který je popsaný v následující podsekci.

Ke zmíněnému obrázku je nutné podotknout, že hesla se v praxi negenerují lexikograficky, tedy tak, jak je na začátku vygenerovaného slovníku zobrazeno. Posloupnost hesel je často závislá na implementaci formátu, délce hesla a také na použití Markovových řetězců, které způsobí generování pravděpodobnějších sekvencí dříve. Pro lepší pochopení struktury masky na obrázku si ukažme znakové sady, které nástroj hashcat standardně nabízí:

- `?l` – zkratka pro *lower*, malá písmena latinky, tedy $a - z$,
- `?u` – zkratka pro *upper*, velká písmena latinky, tedy $A - Z$,
- `?d` – zkratka pro *digit*, arabské číslice, tedy $0 - 9$,
- `?s` – zkratka pro *special*, speciální znaky ASCII, tedy mezera, interpunkce a další,
- `?a` – zkratka pro *all*, zahrnuje všechny předchozí možnosti, tedy `?l?u?d` a `?s`,
- `?h` – zkratka pro *hexa*, znaky šestnáctkové soustavy, tedy $0 - 9$, $a - f$,
- `?H` – zkratka pro *HEXA*, velké znaky šestnáctkové soustavy, tedy $0 - 9$, $A - F$,
- `?b` – zkratka pro *binary*, všechny možné hodnoty znaku, tedy `0x00` – `0xFF`.

Kromě toho je možné do masky zadávat i samotné znaky. Pokud bychom například hledali 8-znakové heslo a věděli, že začíná řetězcem „*pass*“, mohli bychom využít následující masku – `pass?a?a?a?a`. Dále můžeme definovat až 4 vlastní znakové sady. K tomu lze použít buď výčet ASCII znaků nebo výše zmíněné standardně definované znakové sady.

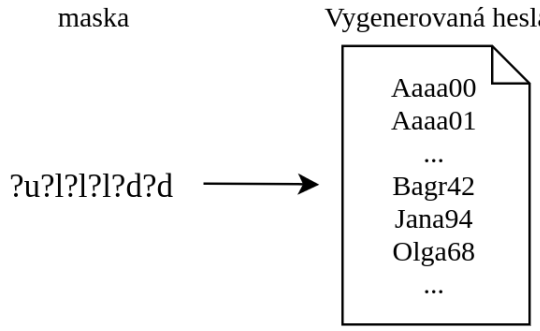
Pokud chceme zadávat Unicode znaky, můžeme využít hexadecimálního zápisu. Zde však narážíme na problém, protože Unicode znaky jsou většinou kódovány dvěma a více bajty. Maska však pracuje na úrovni jednotlivých znaků – bajtů. Pro zadání Unicode znakové sady tedy musíme využít minimálně dva ze čtyř nabízených uživatelsky definovaných znakových sad. Pokud bychom chtěli v jednom hesle využít více takových sad, narazili bychom na limity nástroje hashcat. V praxi to však obvykle problém není, protože nepředpokládáme, že bychom v jednom hesle užili více Unicode znakových sad. Pokud takovou potřebu máme, musíme provést více útoků.

3.1.4 Hybridní útoky

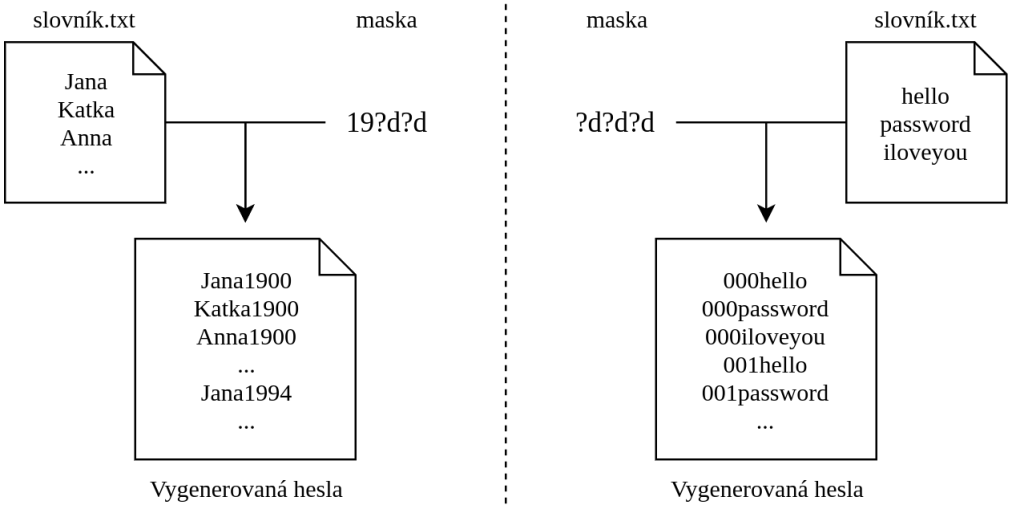
Hybridní útok se v nástroji hashcat vyskytuje ve dvou variantách, označených typem 6 a 7, které jsou k sobě zrcadlové. V hybridním útoku číslo 6 je možné zadat na levé straně slovník s hesly, ke kterému se připojí hesla vygenerovaná maskou, zadané na straně pravé. U hybridního útoku číslo 7 jsou strany prohozené, tedy k vygenerovaným heslům pomocí masky se připojí hesla zadaná ve slovníku.

I když tento útok může být užitečný, je nutné brát v potaz jeho často velmi vysokou časovou složitost – kombinuje totiž dva předchozí útoky a to útok slovníkový a maskový. To bude také problém v následném návrhu plánování, probíraném v kapitole 5.8.

Je také vhodné zmínit, že tento typ útoku je možné nahradit použitím hashcat pravidel (viz sekci 3.3) a to tak, že na každém řádku použitého souboru pravidel umístíme jednu



Obrázek 3.2: Ukázka funkčnosti maskového útoku



Obrázek 3.3: Vlevo ukázka funkčnosti hybridního útoku typu 6, vpravo typu 7

variantu použité masky. V praxi lze použít nástroj *maskprocessor*, dostupný v repozitáři nástroje hashcat, který dokáže vygenerovat na základě masky odpovídající pravidla.

Názornou ukázku generování hesel v hybridních útocích je možné vidět na obrázku 3.3 a to obou typů. Můžeme si všimnou, že hybridní útok číslo 6 generuje podobné výsledky jako příklad z předchozí podsekce o maskách, ale mnohem efektivněji. Do slovníku totiž můžeme uvést pouze seznam jmen a nemusíme se spoléhat na náhodně generované řetězce. Tím rapidně snížíme stavový prostor vygenerovaných hesel. Na straně pravé jsme upravili masku tak, aby nám generovala roky narození a to v rozmezí let 1900 – 1999. Tato maska `19?d?d` nám zřejmě vygeneruje stejný počet hesel jako předchozí maska `?d?d`, protože první dvě číslice jsou pouze statický prefix.

3.2 Generování hesel Markovovými řetězci

Pokud máme zadanou jistou masku, nejnaivnější přístup generování hesel je lexikografické uspořádání. Měnili bychom tedy vždy znak nejvíce napravo a po vyzkoušení všech kombinací bychom zvýšili hodnotu druhého nejpravějšího znaku o jedna, atd. Takto vygenerovaný slovník však nereflektuje pravděpodobnost správnosti jednotlivých hesel. Pokud máme velké množství hesel, může být výhodné vyzkoušet pravděpodobnější hesla dříve.

ϵ	n	p	s	...
a	n	l	t	...
b	o	e	a	...
c	h	i	e	...
⋮	⋮	⋮	⋮	⋱
z	a	e	o	...

Tabulka 3.1: Ukázka matice nahrazující pravděpodobnostní přechody

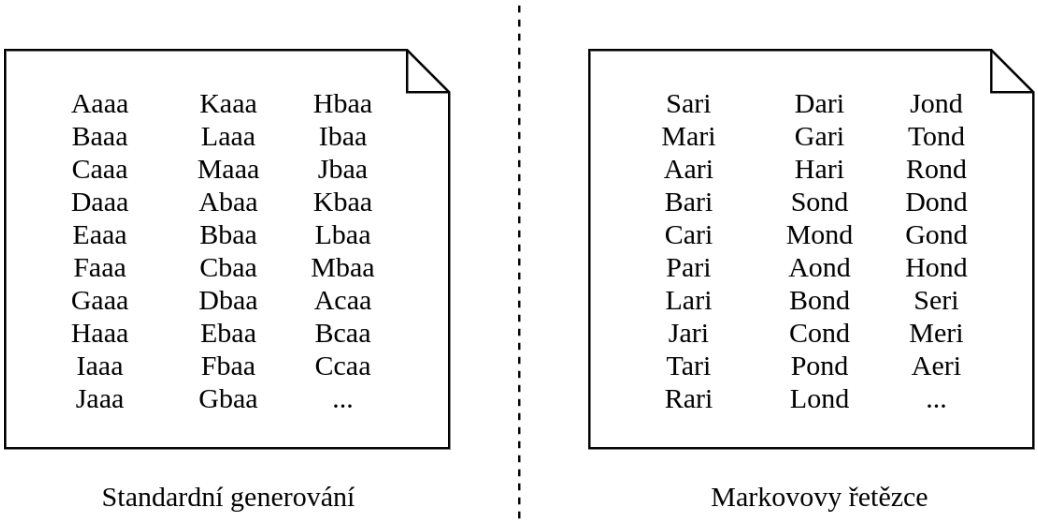
Jedním z nástrojů pro dosažení tohoto cíle jsou Markovovy řetězce. Ty obecně popisují jistý proces, u něhož dochází ke změně stavu s určitou pravděpodobností, založenou na znalosti stavu aktuálního. I když je tato metoda známá již od počátku 20. století, pro potřeby generování hesel byla poprvé navržena v roce 2005 [15]. Pokud máme k dispozici slovník reálných hesel, můžeme určit pravděpodobnost, se kterou bude po znaku `X`, což je aktuální stav, následovat znak `Y`. Tyto pravděpodobnosti následně seřadíme a pro každý znak zapíšeme do matice. První řádek udává pravděpodobnosti prvního znaku v hesle, poté znaku následujícího za znakem `a`, `b` až `z`. Příklad takové matice je možné vidět v tabulce 3.1. Tuto 2D matici je možné dále rozšířit do 3D prostoru pomocí vrstvového modelu, kde matice v každé vrstvě reprezentuje jednu pozici znaku v hesle. Tedy například první sloupec a třetí řádek v páté vrstvě matice by udával nejpravděpodobnější znak následující po znaku *b*, které se nachází v hesle na páté pozici. Pokud máme k dispozici dostatečně velkou trénovací množinu hesel, vrstvový model vykazuje o něco lepší úspěšnost než model klasický [5].

Kromě pořadí vygenerovaných hesel nám Markovovy řetězce dovolují rovněž omezit stavový prostor hesel zvolením hodnoty prahu, tzv. *threshold*. Ten nám určí maximální index sloupce matice, který bude při generování uvažován. Tím dosáhneme vygenerování jen těch nejpravděpodobnějších hesel, relativně k trénovací množině. Bylo ukázáno, že s využitím Markovových řetězců je možné výrazně zmenšit stavový prostor hesel se zachováním velmi vysoké úspěšnosti při hádání hesel [5]. Na obrázku 3.4 je možné vidět porovnání začátku slovníku, vygenerovaného pomocí masky `?u?l?l?l`. Vlevo je generování standardním způsobem nástroje hashcat, který se vzdáleně podobá opačnému lexikografickému uspořádání. Napravo je pak vidět použití Markovových řetězců, které byly natrénovány na známém slovníku hesel *rockyou.txt*, který obsahuje asi 15 milionů hesel.

3.3 Použití pravidel

Pomocí pravidel můžeme provést nejsložitější, ale potenciálně také nejefektivnější útok. Do separátního souboru uvedeme na samostatné řádky pravidla, která upravují slovník ve slovníkovém útoku přesně daným způsobem. Každý řádek může obsahovat více pravidel, která jsou postupně aplikována na všechna hesla slovníku, až získáme výsledný modifikovaný slovník. Funkčnost jednotlivých pravidel by se dala vyjádřit také pomocí regulárních výrazů, ale jejich použití by bylo velmi pomalé a nedostatečné pro potřeby nástroje hashcat. Ten totiž potřebuje často miliardy kandidátů za zlomek sekundy.

Důvodem složitosti útoku je fakt, že syntaxe pravidel je podobná programovacímu jazyku a pro standardní uživatele není jednoduché daný soubor pravidel vytvořit. Tento útok je velmi efektivní z toho důvodu, že nám jednoduše dovoluje ověřit kombinace hesel, které uživatelé volí často, protože jsou dobře zapamatovatelné. Můžeme tedy například k heslu



Obrázek 3.4: Ukázka generovaných hesel pomocí Markovových řetězců

Pravidlo	**Popis**	**Vstup**	**Výstup**
l	Zmenší všechny znaky A–Z	h3sLo	h3slo
C	Zvětší první písmeno a zmenší ostatní	h3sLo	H3slo
t	Změní velká písmena za malá a naopak	h3sLo	H3SlO
r	Převrátí pořadí znaků	h3sLo	oLs3h
]	Smaže poslední znak	h3sLo	h3sL
k	Prohodí první dva znaky	h3sLo	3hsLo

Tabulka 3.2: Ukázka několika hashcat pravidel

připojit znaky na začátek, na konec, prohodit malá a velká písmena a mnoho dalšího. Celkově je možné použít přes 50 různých pravidel a libovolně je řetězit za sebe či je duplikovat a to až do délky 255 funkcí pro jediné heslo. Počet řádků není omezen, je však nutné myslet na fakt, že počet řádků nám násobí velikost stavového prostoru hesel. Výhodou ovšem je, že hesla jsou generována pomocí pravidel až na klientské straně a není je třeba přenášet ze serveru, jak uvidíme při návrhu tohoto útoku v kapitole 5.10.

V tabulce 3.2 je možné vidět několik základních pravidel s příklady jejich funkčnosti. Vzhledem k užitečnosti takových pravidel je navíc jejich syntaxe standardizovaná a lze je použít i v dalších nástrojích pro obnovu hesel jako je například *John the Ripper*. Nástroj hashcat však implementuje část svých vlastních pravidel jako nástavbu.

Pokud chceme vytvořit soubor pravidel, je velmi důležité mít představu, čeho přesně chceme dosáhnout. To často vyžaduje znalost lidského uvažování a častých vzorů v heslech. Následně můžeme pravidla vytvořit samy nebo použít nástroj *maskprocessor* popsaný v následující kapitole 3.4. Ten umí automaticky generovat pravidla na základě zadaných masek. Tímto způsobem tak můžeme napodobit výše popsaný hybridní útok. Máme zde ale mnohem více možností, protože nejsme omezeni pozicí masky za, respektive před hesly slovníku.

Pokud ale nemáme dostatečné znalosti k vytvoření správných pravidel nebo jsme již vyčerpali všechny nápady, můžeme použít speciální schopnost nástroje hashcat generovat náhodná pravidla.

3.4 Podpůrné nástroje

Jak již bylo několikrát zmíněno výše, společně s nástrojem hashcat je k dispozici řada programů, které nám ulehčují práci v mnoha směrech, případně poskytují jisté schopnosti nástroje hashcat jako samostatné aplikace. Tyto nástroje jsou určeny pro pokročilé způsoby obnovy hesel. Většina z nich také pracuje se standardním vstupem a výstupem a je tedy možné je řetězit za sebe. Všechny tyto nástroje jsou vydány pod open-source licencí MIT[4].

V následujících bodech budou popsány některé zajímavé příklady těchto podpůrných nástrojů, které bude možné využít v našem řešení. Celkově je však k dispozici přes 30 samostatných programů.

- **combinator** – Tento program očekává na vstupu dva slovníky hesel, které zkombinuje do jednoho slovníku. Je tak možné převést kombinační útok do klasického útoku slovníkového. Podobnou funkcionalitu má nástroj **combinator3**, který kombinuje 3 slovníky.
- **generate-rules** – Jedná se o samostatnou implementaci generátoru náhodných pravidel, který byl zmíněn výše. Místo volby maximální a minimální délky jednoho pravidla je zde zadáván *seed* v podobě náhodného čísla.
- **hcstatgen** – Nástroj pro generování Markovových statistik ze vstupního slovníku. Tyto pak mohou být použity pro generování dalších slovníků nebo pro samotný útok.
- **keyspace** – Tento nástroj vypočítá stavový prostor hesel ve smyslu hashcat indexů a to ze zadané masky a formátu vstupu.
- **permute** – Jedná se o samostatnou implementaci permutačního útoku. Na výstup jsou tedy vypsány všechny permutace všech hesel vstupního slovníku.

Kromě toho jsou nástrojem hashcat poskytovány celkem 4 generátory slovníků, které dokáží generovat hesla podle několika principů. Tyto nástroje mohou být nabídnuty uživatelům, aby si vytvořili své vlastní slovníky, které pak mohou použít.

- **maskprocessor** – Tento nástroj slouží jako výkonný generátor slovníků na základě vstupních masek. Spolu s nimi je možné také definovat vlastní znakové sady, stejně jako v nástroji hashcat.
- **statsprocessor** – Tento generátor je velmi podobný předchozímu, je však možné zadat navíc soubor s Markovovými statistikami.
- **princeprocessor** – Tento nástroj řetězí jednotlivá hesla ze vstupního slovníku podle speciálních pravidel a vytváří tak hesla nová.
- **kwprocessor** – Tento program generuje hesla na základě znalosti rozložení klávesnice. Z toho také zkratka *key-walker*. Uživatelé často vytváří hesla pomocí jistého vzoru na klávesnici. Taková hesla se na první pohled mohou jevit jako zcela náhodná, i když pomocí tohoto přístupu jsou jednoduše prolomitelná.

[4] Zdrojové kódy hashcat-utils zde: https://github.com/hashcat/hashcat-utils/

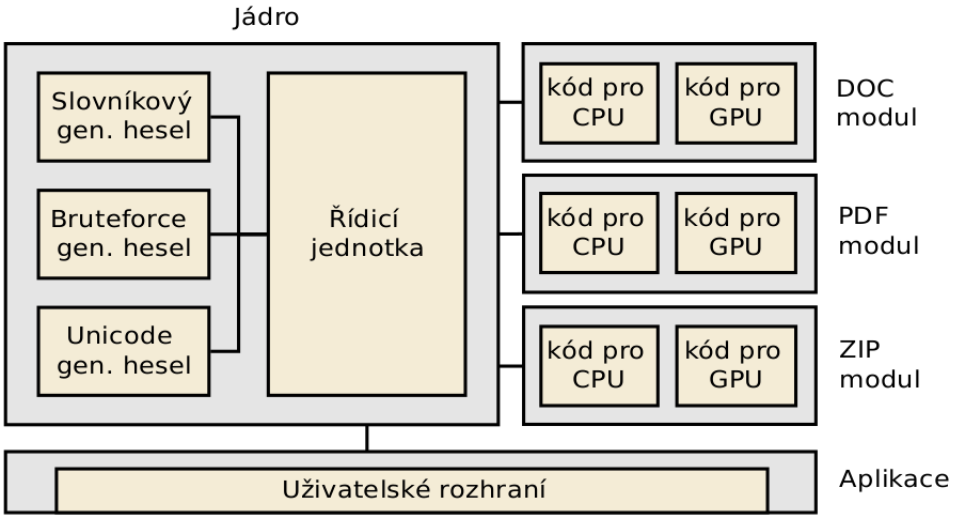
Kapitola 4

Nástroj Fitcrack

Fitcrack je nástroj pro obnovu hesel, jehož vznik byl inspirován nástrojem Wrathion. Ten byl vytvořen v rámci diplomové práce v roce 2014 [17]. Od té doby byl nástroj Fitcrack kompletně přepsán, byla dodána podpora desítek nových formátů, různých znakových sad nebo standardu CUDA. Na tomto samostatném nástroji byla demonstrována možnost značné akcelerace výpočtu s využitím karet GPU a bylo provedeno úspěšné srovnání s tehdejší konkurencí [10].

Nástroj Fitcrack byl vyvinut především pro potřeby forenzní analýzy digitálních dat. V dnešní době se stále častěji setkáváme s médii, která jsou určitým způsobem šifrována. Abychom dokázali získat jejich obsah, musíme znát šifrovací klíč. Pokud jej nemůžeme získat klasickými způsoby, jako je vyzrazení majitelem, jediný způsob je pak získání klíče silou (tzv. *password recovery* nebo *password cracking*).

Samostatný nástroj Fitcrack byl navržen modulárně a přidání podpory nových formátů tak bylo velmi přímočaré. Každý nový modul pak obsahoval kód pro extrakci potřebných informací z daného dokumentu a kód pro CPU a GPU, který prováděl ověřování hesel. Ty byly danému modulu přidělovány pomocí generátorů – například slovníkového nebo *brute-force*. Architektura původního samostatného nástroje je na obrázku 4.1.



Obrázek 4.1: Architektura samostatného nástroje Fitcrack [10]

4.1 Distribuovaný systém

Dalším krokem pro zrychlení procesu obnovy hesel bylo využití více strojů a tedy distribuce výpočtu. Použitá technologie by měla splňovat množství požadavků [11]:

- **dostatečný výkon** - nízké hardwarové nároky pro samotnou distribuci, dostatek výpočetních prostředků pro užitečný výpočet,
- **efektivní výpočet** - minimum času stráveného komunikací a synchronizací uzlů,
- **variabilní počet uzlů** - možnost měnit podobu distribuované sítě i za běhu výpočtu,
- **dobrá škálovatelnost** - co nejblíže lineární,
- **zotavení z chyb** - možnost návratu do konzistentního stavu při výpočetní chybě (např. selhání uzlu),
- **bezpečnost** - možnost nasazení i v nedůvěryhodném prostředí sítě (Internet),
- **jednoduchost použití** - jednoduché nasazení tohoto řešení na výpočetní uzly.

Na základě provedené rešerše různých technologií byl vybrán právě nástroj BOINC, který uspokojivě splňuje všechny uvedené požadavky. Pro správnou funkcionalitu nástroje Fitcrack v distribuovaném prostředí pak bylo potřeba vytvořit tři dodatečné komponenty.

- **Webová administrace** – Jedná se o prostředí, které bude sloužit pro administrátora případně uživatele výpočetního clusteru. V této administraci bude prováděna správa výpočtu a připojených klientů, vytváření nových úloh či zobrazování statistik.
- **Serverová logika** – Toto zahrnuje implementaci komponent systému BOINC jako jsou moduly Generator a Assimilator. Ty zodpovídají za přidělování práce klientům a následné zpracování výsledků. Patří sem také návrh a správa MySQL databáze pro uchovávání potřebných dat.
- **Runner** – Jedná se o middleware pro transformaci požadavků serveru na argumenty lámacího nástroje. Tento program běží na klientských stanicích, sleduje stav výpočtu a poskytuje všechny potřebné informace serveru.

V původní verzi nástroje Fitcrack byly implementovány všechny tyto komponenty. V jednoduchém webovém rozhraní bylo možné přidávat nové lámací balíčky (tzv. *packages*), na kterých se po spuštění podíleli všichni připojení klienti. Dále bylo možné zvolit čas výpočtu dílčích úloh (tzv. *jobs*), které byly následně distribuovány pomocí adaptivního algoritmu. S využitím tohoto systému bylo ukázáno, že efektivita takového distribuovaného řešení roste s náročností obnovovaného souboru [9].

Nicméně, aktuální verze distribuovaného systému Fitcrack je stále prototyp. V rámci projektových praxí v minulých letech bylo do tohoto nástroje doplněno množství funkcí. Můžeme zmínit například schopnost výběru klientů, kteří se budou účastnit na jednotlivých balíčcích nebo grafické rozhraní, které je použitelné jak pro obnovu na jednom počítači, tak jako grafické rozhraní pro klienta, připojeného na server.

4.2 Přechod na hashcat

I když při prvních experimentech na začátku vývoje vykazoval nástroj Fitcrack v porovnání s ostatními nástroji vyšší rychlost, od té doby přešel konkurenční nástroj hashcat do open-source podoby a jeho OpenCL kernely, zodpovědné za ověřování hesel, došly značné optimalizace. Kromě toho podporuje aktuální verze nástroje Fitcrack pouze několik formátů dokumentů, zatímco nástroj hashcat jich v aktuální verzi podporuje přes 200, včetně samostatných hešů, síťových protokolů, operačních systémů, šifrovaných oddílů či kryptoměnových peněženek. S možnostmi výzkumného týmu pracujícím na systému Fitcrack by tento rozsah podporovaných formátů nebylo možné v reálné době zajistit.

Právě kvůli těmto důvodům vznikla myšlenka využití nástroje hashcat jako programu, který bude běžet na klientských stanicích našeho distribuovaného řešení. Získáme tím velmi rychlé řešení obnovy hesel, které bude podporovat obrovskou škálu formátů a navíc bude běžet v distribuovaném prostředí. Abychom tohoto dosáhli, bude nutné přepsat či navrhnout zcela nové komponenty webové administrace, middleware Runner i serverové části systému BOINC a tedy logiku celého našeho řešení.

4.3 Existující řešení

Samotný nástroj hashcat nepodporuje distribuované řešení. Umí využít pouze více zařízení na jednom počítači, jako například více grafických karet. Jeho výhodou je však možnost rozdělení obnovovací úlohy do více částí, a to argumenty `--skip` a `--limit`. Díky tomu je právě nástroj hashcat vhodný pro použití v distribuovaném prostředí na straně klienta.

Distribuovaná hashcat řešení již existují. Většinou se s nimi ale pojí množství nevýhod, jako je neintuitivní ovládání, omezené možnosti využití nástroje hashcat nebo nedostupnost tohoto řešení z důvodu komerční licence. Zde uvádíme příklady těchto systémů s krátkým popisem.

- **Hashtopus**[1] – Jedná se o jeden z prvních veřejně dostupných pokusů o vytvoření distribuovaného řešení, postaveného na nástroji hashcat. Tento systém je řízen serverem s velmi stručným webovým rozhraním, ke kterému je možné přiřadit výpočetní uzly. Nevýhodou tohoto řešení je velmi zastaralé rozhraní pro řízení, klientská aplikace pouze v konzoli a registrace klientů do úlohy pomocí náhodně generovaného tokenu.
- **Hashtopolis**[2] – Toto řešení vychází z nástroje Hashtopus a ve všech směrech ho vylepšuje. Nabízí moderní grafické rozhraní webové administrace, správu klientů a několik úrovní oprávnění. Nevýhodou je klientská aplikace stále ve formě konzole a nutnost vytvářet útoky z velké části ručně, pomocí argumentů příkazové řádky nástroje hashcat.
- **Hashstack**[3] – Jedná se o komerční řešení, které je dostupné pouze při koupi výpočetních stanic od společnosti Sagitta. Z toho důvodu není možné porovnat jeho funkcionality s konkurenčními nástroji. Podle dostupných informací se však jedná o velmi výkonný nástroj, který může díky použití pouze na specializovaném hardware jednoduše předčit ostatní systémy. Kromě toho poskytuje také přívětivé grafické rozhraní.

[1] https://github.com/curlyboi/hashtopus
[2] https://github.com/s3inlc/hashtopolis/
[3] https://sagitta.pw/software/

Kapitola 5

Návrh úprav nástroje Fitcrack

V této kapitole budou navrženy úpravy nutné pro kompatibilitu nástroje Fitcrack s nástrojem hashcat. Kromě serverové části, která je předmětem této práce, bude nutné vytvořit zcela novou verzi programu Runner, který bude na klientské části přijímat pokyny serveru a transformovat je na argumenty nástroje hashcat. Následně bude také zpracovávat jeho výsledky a odesílat je opět na server ve správném formátu. Další částí bude návrh nové webové administrace, která umožní správcům projektu zadávat všechny útoky analyzované v kapitole 3 a která bude využívat dodatečné funkcionality navržené níže. Jak již bylo zmíněno, tyto úpravu budou provedeny mimo tuto práci.

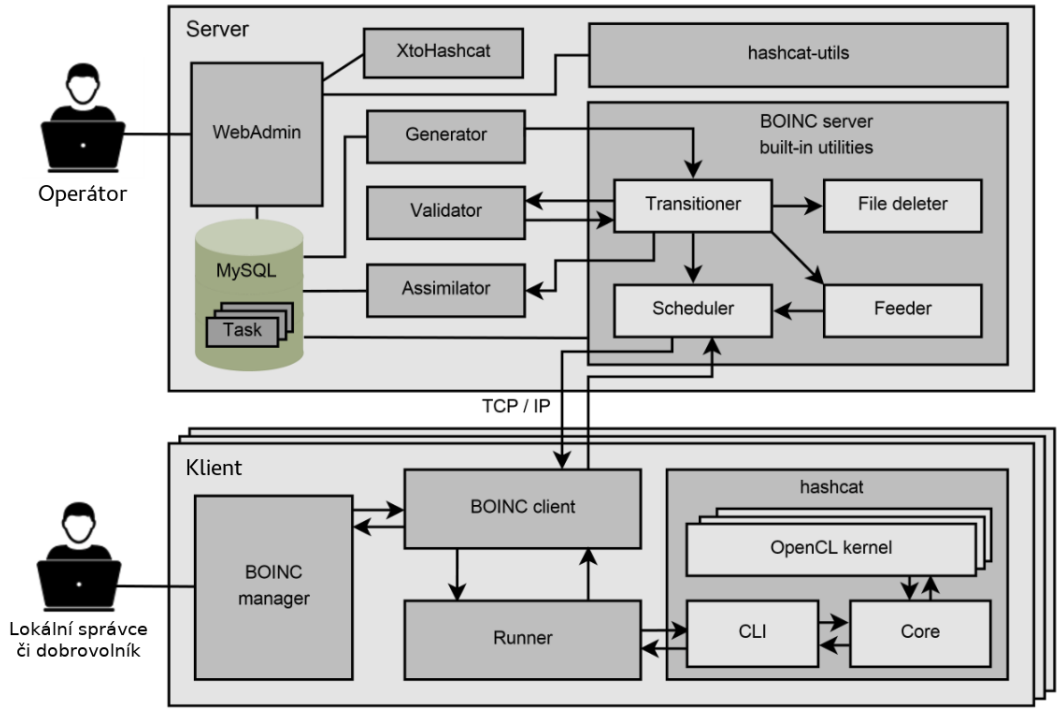
Zde se zaměříme především na serverovou část a tedy strategie plánování jednotlivých útoků a logiku přidělování práce jednotlivým připojeným klientům. Hlavní část práce bude na modulu Generator, který má na starosti právě tyto akce. Bude nutné vytvořit samostatné BOINC šablony pro každý z navržených útoků, abychom klientům mohli zaslat vždy odpovídající vstupní soubory. Dále bude nutné přepracovat program Assimilator, který zpracovává příchozí výsledky od klientů. S těmito úpravami souvisí také nutnost vytvořit zcela nové schéma relační MySQL databáze, kam budou ukládána potřebná data.

Kromě těchto základních úprav jsou kladeny požadavky na dodatečné funkcionality nástroje Fitcrack, které dosud nejsou podporovány. Mezi ně řadíme například automatické měření výkonu klienta ihned po jeho prvním připojení a to všech podporovaných formátů. Tyto informace budou uloženy do samostatné tabulky. Díky nim pak můžeme při zadávání útoku odhadnout čas, který tento útok zabere při volbě různých kombinací klientů. Další požadavek je na automatizaci extrakce potřebných informací z podporovaných dokumentů. Navržený nástroj by měl na vstupu dostat zašifrovaný dokument a na základě něj dát uživateli informaci, o jaký typ souboru se jedná relativně k nástroji hashcat společně s odpovídajícím vstupem. Další požadavky zahrnují například podporu zastavení či restartování běhu balíčků a znovupřidělení nedokončených úloh nebo úloh, které selhaly.

Plánovaná struktura kompletního systému Fitcrack, po dodání funkcionality pro využití nástroje hashcat, je vidět na obrázku 5.1.

5.1 Úpravy modulu Generator

Hlavním místem prováděných úprav pro kompatibilitu s nástrojem hashcat bude modul Generator. V něm probíhá veškeré plánování, vytváření a přiřazování úloh. Sem tedy budou implementovány strategie pro generování jednotlivých útoků popsaných v kapitole 3.



Obrázek 5.1: Plánovaná struktura nástroje Fitcrack [11]

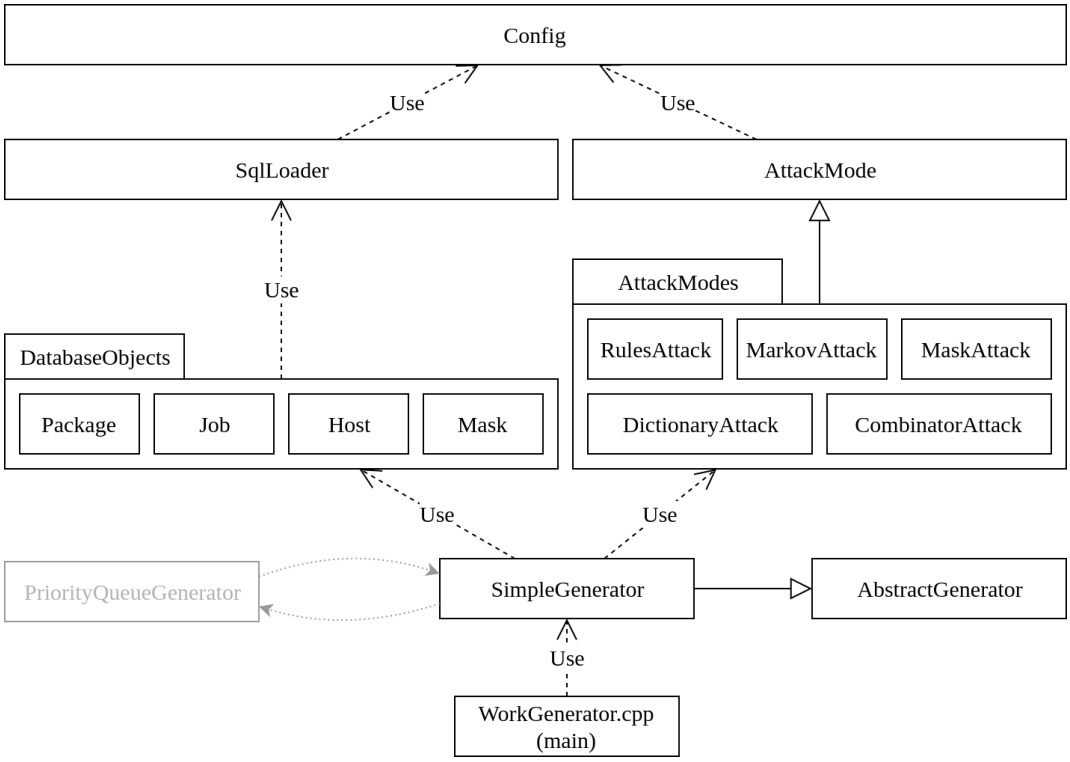
Aktuálně je modul Generator implementován v jednom zdrojovém souboru, který má téměř 2000 řádků a jakékoliv úpravy jsou tak velmi složité, nepřehledné a frustrující. Kromě toho Generator dynamicky vytváří objekty načtené z databáze a následně je nikde neuvolňuje. Tím vznikají úniky paměti, které by bylo vhodné vyřešit. Prvním krokem tedy bude celková refaktorizace modulu Generator.

Modul Generator bude navržen objektově a rozdělen do množství souborů. Díky tomuto návrhu bude jeho správa jednodušší a navíc nám to v budoucnu umožní využít tuto modulární architekturu. Bez obtíží tak bude možné využívat více verzí modulu Generator, které například podporují jiné strategie generování, jiné typy útoků apod.

5.1.1 Objektový návrh

Základní třída, nazvěme ji *CSimpleGenerator*, bude mít na starosti strategii vytváření balíčků. Aktuálně funguje velmi jednoduše – sekvenčně prochází všechny běžící balíčky a vytváří dílčí úlohy pro všechny přiřazené klienty. Pokud běží v celém systému jediná úloha, je tento přístup zcela v pořádku. Pokud by ovšem existoval požadavek na běh více obnovovacích úloh na jednom uzlu, kde každá by měla navíc svoji prioritu, je tento přístup zcela nevyhovující. Výměnou tohoto modulu tedy můžeme dosáhnout změny strategie generování úloh se zachováním všech ostatních komponent systému.

Těchto komponent je několik druhů. Prvně jsou to objekty načítané z databáze – tedy balíčky (*packages*), dílčí úlohy (*jobs*), připojení klienti (*hosts*) a hashcat masky, přiřazené k balíčkům (*masks*). Informace o těchto entitách využíváme napříč celým plánovacím procesem a je tedy vhodné reprezentovat je dynamicky vytvářeným objektem, namísto opakovaných SQL dotazů. Právě tyto objekty způsobují ve stávajícím řešení úniky paměti. Bylo



Obrázek 5.2: Návrh modulární struktury modulu Generator, šedou barvou je naznačeno možné nahrazení modulu SimpleGenerator za modul s odlišnou plánovací strategií

by možné implementovat jistého správce paměti, který by periodicky objekty mazal, pokud by již nebyly potřeba. Jednodušší a čistější řešení však bude využití tzv. chytrých ukazatelů (*smart pointers*), které byly zavedeny se standardem C++11 a které dokáží toto provádět automaticky v pozadí, bez ponechávání této zodpovědnosti na programátorovi.

Druhou důležitou skupinou tříd budou způsoby útoků (*AttackModes*). Ty budou implementovat plánování a vytváření jednotlivých útoků popsaných v kapitole 3. Je zřejmé, že toto vytváření úloh je nezávislé na strategii přidělování práce jednotlivým uzlům, proto je bude možné zachovat napříč různými strategiemi generování úloh. Jak již bylo zmíněno výše, můžeme díky tomu také jednoduše vytvořit verze modulu Generator, které podporují pouze některé typy útoků.

Další pomocné moduly budou poskytovat přístup ke sdíleným globálním proměnným, jako jsou například názvy tabulek nebo šablon, dále společné rozhraní pro přístup k MySQL databázi nebo zpracování argumentů. Popsaná modulární struktura je zobrazena v diagramu 5.2.

5.1.2 Šablony vstupních souborů

Pro odeslání vstupních souborů ze serveru klientům musíme specifikovat BOINC šablonu. Ta je zaslána před přenosem samotných užitečných dat a informuje klienta o tom, kolik souborů má očekávat, jak se budou jmenovat a případně obsahuje další pokyny (například zda si má klient soubor uchovat i po dokončení dílčí úlohy). V našem případě je nutné vytvořit různé šablony pro všechny základní hashcat útoky. Teoreticky by šlo využít stejnou

Jméno	**Popis**
attack	Název útoku (brute|dict|markov|combinator|rule)
attack_mode	Režim útoku v nástroji hashcat (argument -a)
attack_submode	Režim pod-útoku v nástroji hashcat, například využití pravidel
hash_type	Formát vstupu v nástroji hashcat (argument -m)
name	Jméno balíčku odkud úloha pochází
charset1 – 4	Uživatelem definované znakové sady
left_rule	Pravidlo pro úpravu levého slovníku v kombinačním útoku
right_rule	Pravidlo pro úpravu pravého slovníku v kombinačním útoku
mask	Maska pro maskový útok
start_index	hashcat index, od kterého se generují hesla
hc_keyspace	Počet hashcat indexů, které se mají vyzkoušet
mask_hc_keyspace	Stavový prostor maskového útoku, pro výpočet postupu
mode	Informace o typu úlohy (b|n|a)

Tabulka 5.1: Výčet možných záznamů konfiguračního TLV formátu

šablonu ve více útocích, pokud by se shodoval počet a typ zasílaných dat, tato varianta však nenastala.

Existují však soubory, které jsou pro všechny útoky společné. Prvním takovým jsou samotná vstupní data nástroje hashcat. Může se tedy jednat o heš, jehož původní podobu chceme získat, nebo speciální formát, obsahující všechna potřebná data pro získání hesla. Tento soubor se klasicky zasílá pod názvem *data*.

Dále je nutné zasílat jistá konfigurační data. Ta mohou obsahovat například název úlohy, rozsah indexů pro ověření nebo pokyny k využití grafických karet. Tato data jsou často velmi proměnlivá a taktéž v čase se mohou měnit, například s novým požadavkem na schopnosti systému. Proto byl pro jejich přenos zaveden vlastní formát. Ten má vzhled klasického TLV kódování (*Type-length-value*), doplněného o název položky. Každá informace je tedy uložena na samostatném řádku v následujícím formátu.

$$||| < nazev\_polozky > | < datovy\_typ > | < delka\_hodnoty > | < hodnota > |||$$

Díky tomuto formátu můžeme zasílat všechna konfigurační data v jednom souboru. Pokud chceme přidat další informaci, pouze doplníme zpracování této informace na klientské straně v nástroji Runner. Tento soubor je klasicky pojmenován *config*. Kompletní seznam aktuálně využitých položek tohoto *config* souboru je možné vidět v tabulce 5.1.

Další soubory a jejich obsah je již závislý na typu útoku. Nejjednodušší útok z tohoto hlediska je útok maskový. Ten totiž nevyžaduje již žádné dodatečné soubory – maska a indexy pro danou dílčí úlohu jsou již zahrnuty v konfiguračním TLV formátu. Zasílají se tedy pouze tyto soubory:

- **data** – vstupní data pro nástroj hashcat,
- **config** – konfigurační TLV slovník.

Pro slovníkový útok je zřejmě nutné zaslat ještě slovník hesel k ověření, respektive fragment původního slovníku, viz sekci o návrhu slovníkového útoku 5.5. Klientům jsou tedy v tomto případě zasílány 3 vstupní soubory:

- **data** – vstupní data pro nástroj hashcat,
- **config** – konfigurační TLV slovník,
- **dict1** – seznam hesel k ověření.

Pro slovníkový útok s pravidly musíme doplnit tento výčet o další soubor, kterým je seznam pravidel. Je zřejmé, že tyto pravidla zůstanou po celou dobu výpočtu balíčku neměnné. Je proto vhodné zasílat je s příznakem *sticky*, i když mají velikost maximálně několik stovek kilobajtů. Tento příznak zajistí, že daný soubor na klientské stanici zůstane napříč všemi dílčími úlohami a nebude se přenášet při každé této úloze znovu, jako je tomu u ostatních souborů. Seznam zasílaných souborů je pak následující:

- **data** – vstupní data pro nástroj hashcat,
- **config** – konfigurační TLV slovník,
- **dict1** – seznam hesel k ověření,
- **rules** – seznam pravidel.

Stejný počet souborů jako předchozí útok má i útok kombinační. Místo seznamu pravidel se však zasílá fragment druhého slovník. První slovník je nutné zasílat celý, což je vysvětleno v následující sekci o plánování kombinačního útoku 5.6. Abychom si zbytečně nepletli zasílané soubory, má tento útok svoji vlastní šablonu. Ta definuje přenos následujících souborů:

- **data** – vstupní data pro nástroj hashcat,
- **config** – konfigurační TLV slovník,
- **dict1** – první slovník hesel,
- **dict2** – fragment druhého slovníku.

Posledním útokem a zároveň poslední šablonou je maskový útok s využitím Markovových řetězců. Zde je nutné kromě masky v TLV slovníku zaslat hashcat *hcstat* soubor, který obsahuje informace o postupu generování hesel. Tento soubor je bohužel značně veliký – v aktuální verzi nástroje hashcat 3.6.0 má statickou velikost 32MB. Je tedy vhodné opět využít *sticky* příznak, abychom zabránili opětovnému zasílání mezi dílčími úlohami. Poslední šablona tedy vypadá následovně:

- **data** – vstupní data pro nástroj hashcat,
- **config** – konfigurační TLV slovník,
- **markov** – binární *hcstat* soubor s pravděpodobnostní maticí.

Jak je vidět, počet zasílaných informací ke každé úloze je výrazný. To může zvyšovat režii a je tedy vhodné tvořit relativně delší obnovovací dílčí úlohy. Kromě již několikrát zmíněného *sticky* příznaku je vhodné řešit také zasílání dlouhých slovníků u slovníkového a kombinačního útoku. Zde nemůžeme tento příznak využít, protože obsah zasílaných fragmentů slovníků se s každou úlohou liší. Můžeme tak alespoň provést komprimaci slovníků na serveru, čímž snížíme jejich velikost a čas nutný k přenosu. U *hcstat* souboru to bohužel nebude mít valný význam, vzhledem k jeho binárnímu formátu. Zmíněných 32MB tedy bude nutné přenést v původní velikosti.

5.1.3 Schéma databáze

V databázi MySQL jsou soustředěna všechna potřebná data. Nalezneme zde množství BOINC tabulek, které jsou vygenerovány automaticky při tvorbě projektu. S těmito tabulkami pak pracuje vnitřně systém BOINC a administrátoři projektu je nemodifikují.

Dále jsou zde přítomny námi vytvořené tabulky. Ty obsahují užitečná data, která slouží pro generování úloh, ukládání výsledků výpočtu, nalezených hesel apod. Jedná se o následující tabulky:

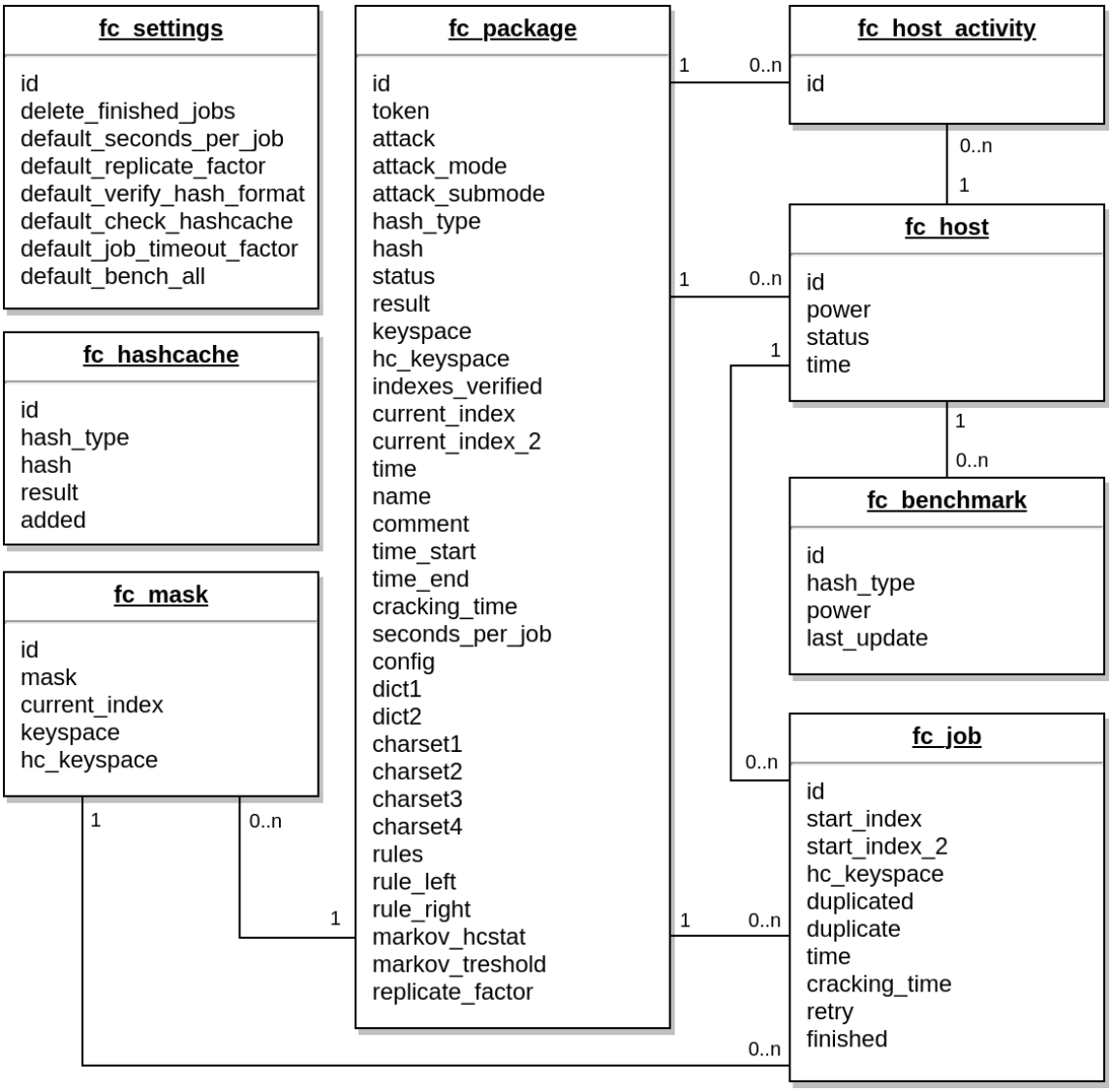
- fc_benchmark,
- fc_hashcache,
- fc_host,
- fc_host_activity,
- fc_job,
- fc_mask,
- fc_package,
- fc_settings.

V původní verzi nástroje Fitcrack již byla MySQL databáze používána. Pro potřeby využití nástroje hashcat ji však bylo nutné upravit a doplnit o další sloupce či celé tabulky. Kompletní schéma spolu s popisem jednotlivých tabulek, sloupců, jejich možných hodnot i komentářem k provedeným úpravám je možné nalézt v příloze B. Vztah jednotlivých tabulek je možné vidět v diagramu 5.3.

5.1.4 Znovupřidělení nedokončených úloh

Kromě vygenerování úloh z celého stavového prostoru je žádoucí, aby byly všechny úlohy korektně dokončeny. V opačném případě by totiž mohlo dojít k selhání dílčí úlohy, v jejímž stavovém prostoru se právě heslo nacházelo. Pokud tedy úloha selže, Generator ji musí přidělit jiným uzlům.

Příčiny tohoto selhání mohou být dvě. První je nezaslání výsledku a to buď z důvodu odpojení klienta nebo pádu klientské aplikace. V aktuálním stavu je jedinou možností pro detekci takového selhání *timeout*, tedy vypršení stanovené doby pro výpočet dané dílčí úlohy. Toto řešení je široce využíváno i v ostatních BOINC projektech, založených na dobrovolnickém výpočtu. Otázkou zůstává, jaký *timeout* zvolit. Aktuální verze benchmarků nástroje hashcat poskytuje jen velmi hrubé orientační doby výpočtu, které se tak mohou výrazně lišit od reálné doby výpočtu. Nechceme tedy odepsat uzel, který má již téměř hotový výpočet, jako odpojený. Na druhou stranu, vysoká hodnota *timeout* nám může zbytečně zpomalovat celkovou dobu výpočtu, pokud k tomuto selhání dojde. Tento čas je možné zvolit uživatelem v MySQL tabulce *fc_settings* ve sloupci *default_job_timeout_factor* a to jako násobek původně plánované doby výpočtu dílčí úlohy. Další možností pro řešení nezaslání výsledku by bylo zavést periodickou komunikaci mezi klientem a serverem v průběhu výpočtu. Zde by navíc mohl být zasílán aktuální postup ve výpočtu úlohy. Pokud by došlo k zastavení této komunikace, mohl by být klient prohlášen za odpojeného a úloha přidělena někomu jinému, bez zdlouhavého čekání. Tento přístup by však vyžadoval návrh a implementaci nového BOINC démona Trickler, který je určen právě k tomuto účelu. To je však již nad rámec této práce a tento postup tedy bude možné využít při budoucím výzkumu.



Obrázek 5.3: UML diagram zobrazující vztahy mezi tabulkami databáze

Druhou příčinou selhání mohou být například chybějící ovladače, nepodporovaný hardware nebo pouze přechodný nedostatek výkonu. Obecně tedy chyby, které klientská strana detekuje a serveru zašle informaci o selhání výpočtu bez zdržení.

V době obdržení zprávy o selhání může být aktuální index stavového prostoru pro generování nových úloh zcela jinde a bylo by nutné generovat již dokončené úlohy zcela znovu, což by bylo velmi neefektivní. Místo toho je nedokončené úloha označena příznakem *retry*. Díky tomu, že si úloha uchovává informace o své velikosti a poloze ve stavovém prostoru hesel, je možné ji využít pro generování nových úloh. Protože velikost každé vygenerované úlohy je připravena přesně pro aktuální výpočetní sílu daného stroje, kopírování celé úlohy do nové není zcela efektivní. Mohlo by se tak stát, že by úlohu, původně přidělenou velmi výkonnému stroji v našem výpočetním clustru, nově dostal nevýkonný notebook, který by si s ní nedokázal v rozumném čase poradit. Proto bude v těchto případech zavedeno fragmentování nedokončené úlohy. Původní úloha bude rozdělována postupně. Pokud však o danou úlohu bude mít zájem uzel s podobnou výpočetní silou, bude mu přidělena celá i za cenu menšího zdržení nebo zrychlení výpočtu. Přidělování miniaturního zbytku by totiž mělo mnohem větší dopad na výkon, z důvodu zbytečné režie a následného výpočtu v řádu sekund.

Modul Generator tedy před vygenerováním nových úloh vždy zkontroluje, zda se v systému nachází nedokončené úlohy. Pokud ano, provede generování nových úloh právě z těchto úloh. V opačném případě teprve přejde ke generování úloh s novými indexy.

5.1.5 Zastavení a restart výpočtu

V původní verzi nástroje Fitcrack neexistoval způsob, jakým by uživatel mohl zastavit výpočet. Ten byl tak vždy dokončen buď kladně, kdy bylo nalezeno heslo, nebo s výsledkem negativním, kdy byl vyčerpán stavový prostor. Zřejmě bychom ale chtěli mít možnost výpočet zastavit, případně pozastavit a následně v něm pokračovat.

Pokud chceme úlohu restartovat, prvním požadavkem je, aby uzly, které aktuálně úlohu provádí, byly zastaveny. Pak můžeme nastavit klíčové položky balíčku, jako je aktuální index stavového prostoru a počet již ověřených hesel na výchozí hodnoty a balíček spustit znovu. Nicméně operace restartu daného balíčku je určena spíše pro testovací účely. V praxi by nemělo smysl ověřovat znovu již jednou ověřená hesla.

Důležitým problémem je tedy samotné zastavení, respektive pozastavení daného distribuovaného výpočtu. Nejjednodušším řešením, které podporují i všechny konkurenční nástroje zmíněné výše v sekci 4.3, je zastavení generování nových úloh. To je možné provést jednoduše uvedením balíčku do stavu 12 (*Finishing*) v naší databázi. V tomto případě dojde k dokončení všech naplánovaných úloh. Následně je balíček automaticky uveden do stavu 0 (*Ready*) a je tak efektivně pozastaven. To znamená, že může být kdykoliv znovu spuštěn a výpočet bude pokračovat tam, kde skončil. Nevýhodou je čekací doba, kdy se dokončují všechny naplánované úlohy. Ta může být aktuálně dlouhá až $2 \times n$, kde n je plánovaný čas jedné dílčí úlohy a to z důvodu, že v databázi se vždy udržují 2 naplánované úlohy, pro minimalizaci režie.

Pokud bychom chtěli provést okamžité zastavení, je možné využít schopnosti systému BOINC zaslání příkazu ke stornování úloh všem klientům. V takovém případě ale nezískáme výsledky aktuálně prováděných ani naplánovaných výpočtů. Tím tedy uvedeme plánovač do nekonzistentního stavu. Ten má totiž nastaven aktuální index stavového prostoru na hodnotu, která je větší, než reálně provedené výpočty. Můžeme ho však ale uvést do výchozího stavu a výpočet restartovat. Druhou možností by bylo označit všechny stornované a tedy

nedokončené úlohy v naší databázi příznakem *retry*. Tím bychom dosáhli toho, že při dalším spuštění by byly vygenerovány nejprve stornované úlohy, respektive úlohy z dosud neověřeného stavového prostoru. Tímto způsobem bychom navrátili plánovač do konzistentního stavu. Dosáhneme tak tedy okamžitého zastavení distribuovaného výpočtu s možností následně pokračovat tam, kde byla úspěšně dokončena poslední dílčí úloha. Nevýhodou může být zvýšená režie z důvodu nutnosti fragmentace těchto nedokončených úloh, jak bylo popsáno v předchozí podsekci 5.1.4. Tuto schopnost ale žádný ze zmíněných konkurenčních nástrojů nepodporuje. Bude tedy záležet na požadavcích výsledného systému, která z popsaných metod zastavení bude ve výsledku implementována.

5.2 Úpravy modulu Assimilator

Druhou částí našeho serverového řešení, které bude nutné upravit, je modul Assimilator. Jak již bylo zmíněno výše, jedná se o BOINC démona, který periodicky kontroluje existenci nových validních výsledků výpočtu. Tento program byl již částečně funkční i v původním řešení, ovšem pro potřeby využití nástroje hashcat bude nutné jeho funkcionalitu upravit a přidat nové vlastnosti.

Jednou ze zcela nových schopností nástroje Assimilator je ukládání nalezených hesel do samostatné tabulky *fc_hashcache*. Ta pak obsahuje kombinaci původního hashcat vstupu a odpovídajícího nalezeného hesla. Tyto informace lze následně využít při vytváření nových balíčků – pokud je ke hledanému vstupu nalezen výsledek již v této tabulce, nemusí být vůbec prováděn výpočet.

I když je modul Assimilator implementován podobně nepřehledně, jako v případě modulu Generator, jeho funkcionalita je mnohem jasnější. Pouze zpracovává příchozí výsledky a nepředpokládáme, že by se toto výrazně měnilo s nadcházejícími verzemi. Navíc zde máme jako správci projektu svázané ruce, protože implementace modulu Assimilator spočívá pouze v implementaci jediné funkce. Ta je následně volána z nekonečné smyčky, implementované vývojáři systému BOINC. Nelze si tedy jednoduše vytvořit zcela nový projekt, jako tomu bylo u modulu Generator. Z těchto důvodů nebude prováděn kompletně nový návrh, jako tomu bylo výše u modulu Generator, ale bude využita architektura stávající.

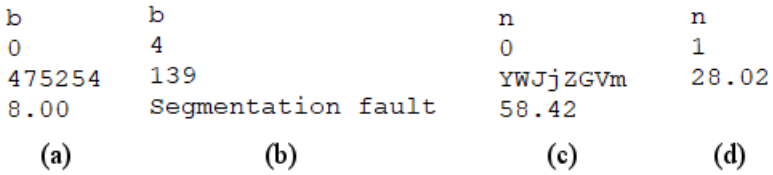
Zasílání výsledků výpočtu

Hlavní zodpovědností modulu Assimilator je zpracování dat, zaslaných směrem z klienta na server. Existují celkem tři typy úloh, jejichž výsledky chceme zpracovat.

- **Benchmark** – Tato úloha měří výkon stanice pro daný formát.
- **Výpočet** – Jedná se o normální výpočetní úlohu, kdy uzel hledá heslo či vstup heše.
- **Bench_all** – Tento typ úlohy měří výkon stroje pro kompletní seznam podporovaných formátů a její funkcionalita je rozebrána v sekci 5.3 níže.

Formát zasílaných zpráv se liší pro každý zmíněny typ výsledku. Na každém řádku je vždy uvedena informace, jejíž obsah pak udává možné hodnoty dalších řádků společně s jejich významem. První dva řádky všech typů výsledku však mají formát společný.

- **<typ výsledku>** – Jedná se o jednoznakovou hodnotu, udávající jeden ze tří zmíněných typů výsledku, tedy *b* pro benchmark, *n* pro normální výpočet a *a* pro úlohu typu *bench_all*.



Obrázek 5.4: Příklady reálně zasílaných výsledků – (a) úspěšný benchmark s naměřenou rychlostí a dobou průběhu, (b) selhání benchmarku s kódem hashcat chyby a popisem, (c) nalezené heslo v kódování base64 s časem výpočtu, (d) heslo nenalezeno s časem výpočtu

- **<stav výsledku>** – Jedná se o celočíselnou hodnotu, která informuje o výsledku. Má následující význam.
  - **0** – V případě benchmarku značí tato hodnota úspěšné naměření výkonu, v případě výpočtu pak úspěšné nalezení hesla.
  - **1** – Tato hodnota je používaná pro informaci o nenalezení hesla ve výpočetní úloze.
  - **(>= 3)** – Tento stavový kód značí výskyt chyby. Zpráva je většinou doprovázena podrobnějším výpisem o chybě na následujících řádcích.

Obsah dalších řádků se pak liší v závislosti na typu výsledku a jeho stavovém kódu. Například při získání hesla je tento nález zaslán na třetím řádku, zakódovaný pomocí base64, dále se při úspěšném výpočtu u všech typů výsledků zasílá čas běhu dané dílčí úlohy. Kompletní komunikační protokol je možné nalézt v příloze C. Dále je přiložen obrázek 5.4 s popisky, ukazující příklady zaslaných odpovědí.

5.3 Automatické měření výkonu

Po připojení uzlu do výpočtu balíčku je vždy proveden nejprve benchmark, který dá serveru orientační informaci o tom, jak velkou úlohu zvládne tento uzel za daný čas spočítat. Po dokončení útoku je však tato informace zahozena a neexistuje tedy globální pohled na celkový výpočetní výkon dané množiny klientů. Ten bychom však chtěli získat. Díky této informaci můžeme aproximovat dobu běhu daného útoku při účasti dané kombinace klientů. Tato informace může být dále použita pro inteligentní metody plánování – můžeme si například vybírat, kterým uzlům přidělíme jakou práci, protože různý hardware může mít rozdílné rychlosti u jednotlivých formátů. Tu samou informaci bychom potřebovali i u plánování prioritních front úloh.

Právě za tímto účelem byla navržena již zmíněná tabulka *fc_benchmark*. Správce projektu bude mít možnost spustit na vybraných klientech kompletní benchmark – měření rychlosti všech podporovaných formátů nástrojem hashcat. Server následně obdrží seznam rychlostí, který si zkopíruje do zmíněné tabulky. I když nástroj hashcat podporuje spuštění právě kompletního benchmarku, tento proces je u jistých formátů velmi náročný na paměť grafické karty, což může v aktuální verzi nástroje hashcat (3.6.0) způsobit pád aplikace, či dokonce celého operačního systému, pokud je benchmark spuštěn v příliš náročném módu. Program Runner, který požadavek na kompletní benchmark obdrží, bude postupně spouštět benchmarky jednotlivých formátů. Díky tomu získáme kompletní seznam i v případě, že se některé benchmarky nepovedou.

Takový kompletní benchmark trvá navíc značnou dobu. V závislosti na počtu grafických karet na klientské stanici a jejich výpočetním výkonu se může jednat o desítky minut až hodiny. Bylo by tedy vhodné toto provádět co nejdříve a následně správce projektu neobtěžovat, až bude tyto informace potřebovat. Z toho důvodu bude proces kompletního měření prováděn automaticky, ihned po připojení nového klienta do systému. Toho bude dosaženo vložením MySQL spouště do naší databáze (tzv. *trigger*). Ten bude využívat faktu, že po připojení neznámého uzlu do projektu je vložen nový záznam s informacemi o tomto novém klientovi do BOINC tabulky *host*. Pokud k tomuto vložení dojde, námi navržená spoušť nastaví speciální balíček na začátku tabulky *fc_package*, s ID rovným hodnotě 1, na stav 10 (*Running*). Díky tomu modul Generator naplánuje novou úlohu, v jejímž konfiguračním souboru bude zaslán požadavek na kompletní benchmark.

5.4 Automatická detekce formátu a extrakce heše

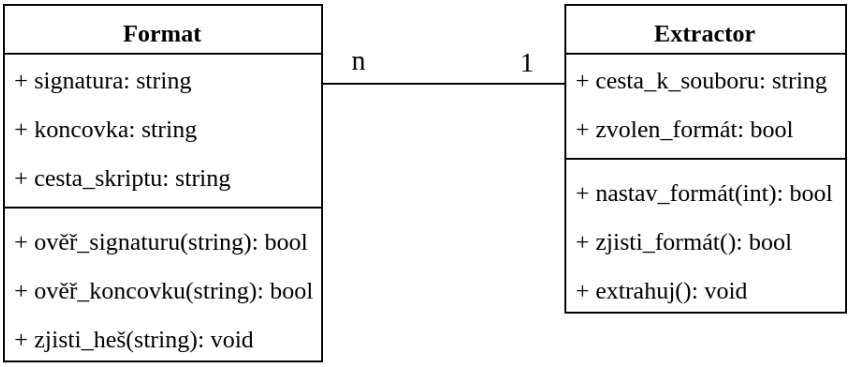
Nástroj hashcat podporuje mimo jiné také obnovu souborů, jako jsou archivy ZIP, RAR či 7-Zip, dokumenty PDF, Microsoft Office a další. Vstup do tohoto nástroje je však velmi specifický pro každý zmíněný formát. Tento vstup obsahuje všechny informace nutné pro ověření a tedy i získání hesla daného souboru. Extrakce těchto dat však často není zcela triviální a je nutné pracovat s textovou či binární strukturou jednotlivých formátů, která se navíc může lišit s každou novější verzí. Pokud tedy máme zašifrovaný soubor, musíme z něj nejprve získat potřebné informace.

K tomu můžeme využít existující veřejně dostupné skripty v nejrůznějších jazycích, které zvládnou právě tuto extrakci. I když použijeme spolehlivý program, ani po extrakci přesně nevíme, jaký typ heše vlastně máme. Například pro dokumenty Office existuje extrakční skript, který poskytne vstup do nástroje hashcat. Následně však ale nevíme, o kterou verzi Office se jedná a jaké číslo formátu do nástroje hashcat tedy zadat.

Cílem je vytvořit vlastní nástroj, který využívá pokud možno spolehlivých, volně dostupných skriptů. Na vstupu přijme libovolný zašifrovaný soubor, podporovaný nástrojem hashcat, podle jeho signatury a koncovky rozpozná, o jaký formát se jedná a aplikuje na něj vybraný skript. Analýzou získaného výsledku navíc zjistí, o kterou verzi daného dokumentu se jedná. Tyto dvě informace následně vytiskne na samostatné řádky. Ty jsou pak dostačující pro zjištění identity šifrovaného souboru a můžeme pomocí nich spustit obnovu hesla námi definovaným způsobem.

Může se však stát, že dokument má chybnou koncovku nebo se jedná o soubor, který z principu být identifikován ani nemůže – například šifrovaný oddíl TrueCrypt. Pokud však uživatel ví, nebo má podezření, o který formát se jedná, musíme mu umožnit využít extrakčních skriptů. Z toho důvodu bude mít navržený nástroj, nazvěme ho podle jeho funkcionality XtoHashcat, argument `-f <číslo_formátu>`, kterým bude možné ručně specifikovat formát. Následně proběhne pokus o spuštění daného extrakčního skriptu. Uživatel nebude muset zadávat přesnou verzi dokumentu – stačí pouze informace o tom, zda se jedná například o Office, či PDF a exaktní verze je pak vypsána analýzou získaného výstupu, pokud je extrakce úspěšná.

Nástroj XtoHashcat bude navržen objektově. Třída *ArgumentParser* nejprve zajistí zpracování argumentů. Každý formát pak bude reprezentován samostatnou instancí třídy *Format*, jejíž hlavní atributy budou signatura, koncovka a cesta k extrakčnímu skriptu. Hlavní třída *Extractor* při vzniku dynamicky vytvoří instance všech podporovaných formátů a podle zadaných argumentů nastaví formát ručně nebo se pokusí o jeho identifikaci. Pokud je tato fáze úspěšná, spustí *Extractor* skript daného formátu, jehož výstup dále ana-



Obrázek 5.5: Diagram struktury nástroje XtoHashcat

lyzuje a hledá v něm klíčové prvky. Následně je vypsán výstup ze spuštěného skriptu spolu s identifikovanou verzí dokumentu. Diagram popsané architektury je zobrazen na obrázku 5.5.

Nástroj XtoHashcat bude využíván při vytváření nových balíčků v administraci nástroje Fitcrack. Pokud správce nahraje zašifrovaný soubor, bude mít možnost zvolit automatickou detekci formátu bez nutnosti zadávání dalších informací o zaslaném souboru.

5.5 Slovníkový útok

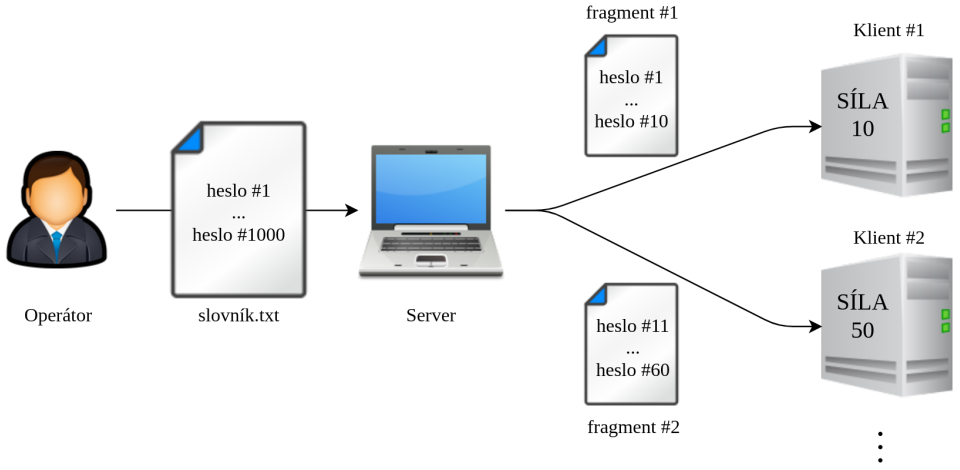
Nyní se již podívejme na návrh samotných hashcat útoků a jejich provedení v distribuovaném prostředí systému BOINC. Prvním takovým útokem je útok slovníkový.

Pro vytvoření úlohy slovníkového útoku musíme každému klientu zaslat seznam hesel, které má ověřit. Například oproti maskovému útoku, kde se zasílá pouze maska a rozsah indexů, je to velký nárůst v zasílaných datech, což se negativně projeví na rychlosti samotné obnovy. V praxi pak záleží na rychlosti přenosu dat, ale také na formátu souboru nebo heše, který se snažíme obnovit. V naší předchozí práci jsme experimentálně ověřili, že složitost obnovovaného formátu je přímo úměrná efektivitě slovníkového útoku [12]. Pokud je totiž složitost nízká a dokážeme ověřovat například stovky miliard hesel za sekundu, zřejmě nebudeme schopni takovou rychlostí danému klientu hesla po síti zasílat. Smysl distribuovaného slovníkového útoku na takový formát se pak ztrácí.

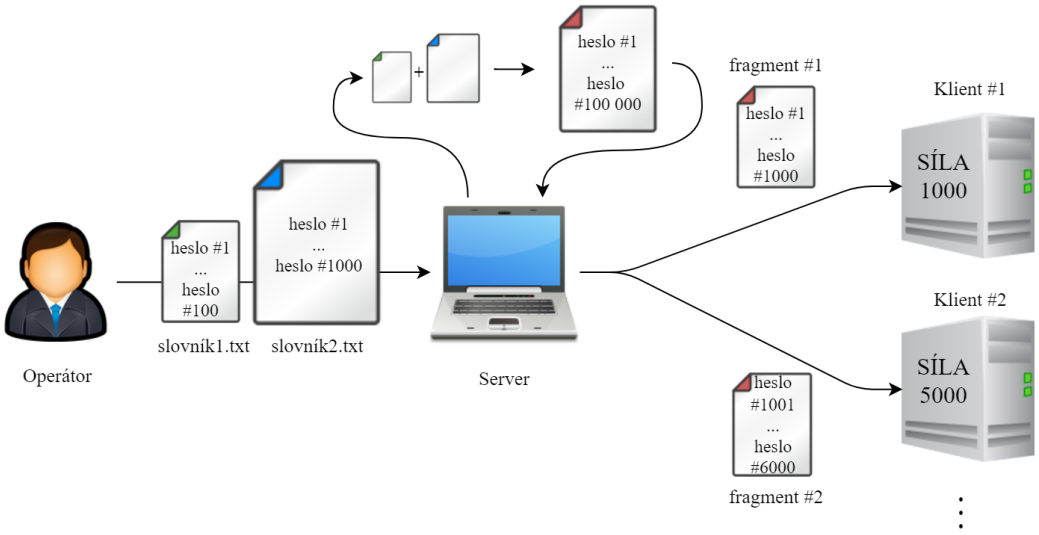
Výhodou slovníkového útoku pomocí nástroje hashcat je fakt, že stavový prostor udávaný tímto nástrojem je roven počtu hesel ve slovníku. Tím nám odpadá složitý přepočet, který je nutné provádět u maskového útoku.

Generování slovníkové úlohy pak bude velmi přímočaré. Slovník, jehož obsah chceme ověřit, je postupně rozdělován do fragmentů, které jsou následně zasílány jednotlivým klientům. Server si pro daný balíček typu slovníkový útok udržuje informaci o posledním přiděleném indexu. Při generování nové úlohy pak přeskočí tento počet hesel, tedy počet řádků ve slovníku, a do fragmentu slovníku zasílaném klientovi zkopíruje daný počet hesel v závislosti na předem změřeném výkonu klienta a požadované době běhu jedné úlohy. Pokud je při kopírování dosaženo konce slovníku, je balíček uveden do stavu 12 (*Finishing*), což značí, že již byla vygenerována všechna práce a brzy dojde k dokončení úlohy, a to buď úspěšným nalezením hesla nebo vyčerpáním stavového prostoru.

Princip generování úloh slovníkového útoku je možné vidět na obrázku 5.6. Kromě toho je zde také zobrazena síla jednotlivých stanic a k nim odpovídající velikost fragmentu.



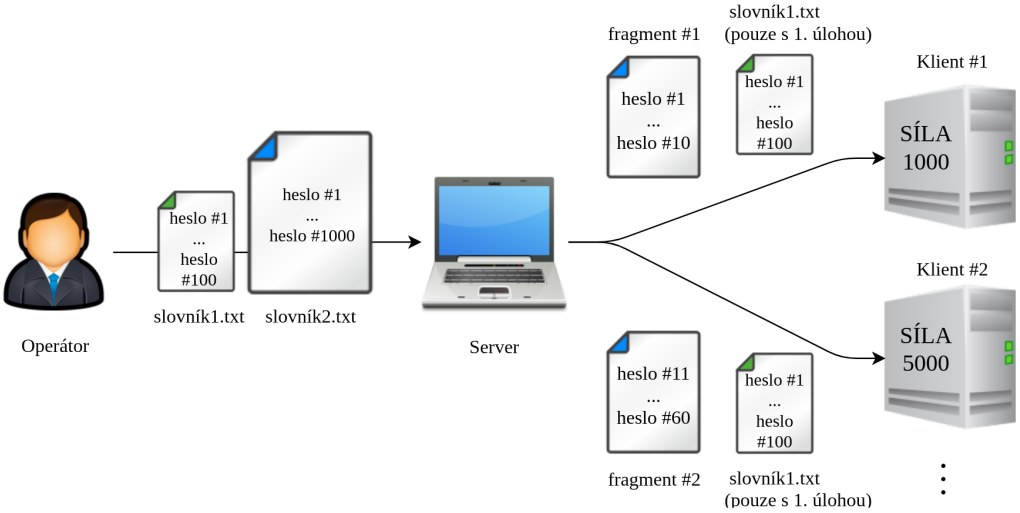
Obrázek 5.6: Distribuce slovníku mezi klienty



Obrázek 5.7: Naivní přístup ke kombinačnímu útoku – generování na straně serveru

5.6 Kombinační útok

Nejnaivnějším přístupem by bylo využití nástroje dostupného v repozitáři *hashcat-utils* nazývaného *combinator*. Tento nástroj funguje jako samostatná implementace kombinačního útoku. Na vstupu očekává 2 slovníky a jeho výstupem je seznam všech možných kombinací hesel z obou slovníků. Následně bychom s takto získaným slovníkem provedli klasický slovníkový útok, popsaný v předchozí kapitole. Tento přístup je však nanejvýš neefektivní, protože by bylo nutné přenášet obrovské množství hesel navíc. U původního kombinačního útoku je potřeba v nejlepším možném případě přenést pouze $(m + n)$ hesel, a to v případě, kdy by celý balíček zvládl vypočítat jediný uzel. Ve výše navrhovaném řešení by to však bylo $(m \times n)$ hesel – prostorová složitost by se tedy zvýšila z lineární na kvadratickou. Takovýto přístup je možné vidět na obrázku 5.7.



Obrázek 5.8: Reálné provedení kombinačního útoku – generování na klientské straně

Slovníky tedy bude nutné určitým způsobem rozdělit již v modulu Generator. Bohužel však není možné fragmentovat oba slovníky zároveň, protože by nedošlo k ověření kombinací hesel z dvou rozdílných úloh. Zbývající možností je tedy zaslat každému výpočetnímu uzlu jeden slovník celý a další slovník poté dělit na jednotlivé fragmenty. Zřejmě zde tedy nedosáhneme ideální lineární prostorové složitosti zmíněné výše, ve většině případů bude ale prostorová složitost drasticky nižší, než pokud bychom posílali již zkombinovaný slovník. Opačný případ by nastal jen tehdy, pokud by klienti vždy vyzkoušeli pouze 1 heslo z druhého slovníku (či méně, viz dále) nebo by bylo připojeno více klientů, než je hesel v zasílaném slovníku, což při běžné velikosti slovníků desítek čí stovek megabajtů v praxi nenastane. Diagram tohoto řešení vidět na obrázku 5.8. Je možné si všimnout právě velké úspory zasílaných dat mezi obrázky 5.7, kde je nutné klientovi o síle 1000 zaslat vždy tisíc hesel, zatímco v přístupu 5.8 je to při každé nové úloze pouze 10 hesel.

Při takovémto přístupu se vyskytují ještě dva zřejmé problémy. Vzhledem k tomu, že první zaslaný slovník se s časem nemění, není nutné jej zasílat všem klientům při každé přidělené úloze znovu, tedy tak, jak to probíhá u typických projektových souborů. To by způsobilo obrovské zpomalení celého procesu, a to kvůli již zmíněné velmi značné velikosti slovníků. Řešením je využití tzv. *sticky* příznaku ve vstupní šabloně útoku, kterým klientům sdělíme, aby daný soubor po dokončení úlohy nemazali, ale nechali si jej pro další práci. Navíc je nutné si informaci o zaslání pamatovat na serveru. Toto však již systém BOINC řeší sám. Modul Generator tedy naplánuje odeslání všech souborů. Pokud se ale již na klientu nachází soubor se stejným názvem a obsahem, k opětovnému přenosu nedojde.

Druhým problémem je minimální počet hesel, který je možné v jedné úloze vyzkoušet. Ve výše navrženém postupu by to bylo minimálně $(n \times 1)$ hesel. Pokud však bude n příliš velké nebo rychlost obnovy velmi malá, může to mít za následek mnohem delší dobu výpočtu jedné úlohy, než požadoval uživatel. Taková situace by mohla v praxi jednoduše nastat, protože velikost reálných slovníků může být v jednotkách milionů hesel a ještě mnohem více, pokud by šlo o slovník vygenerovaný na základě pravděpodobností, gramatik [18] či jiných technik [8]. Ty totiž často dovolují generovat předem neomezené množství hesel a rovněž nástroj hashcat není nijak limitován velikostí slovníků. Řešením je zavedení nového sloupce v tabulce *fc_package – hc_index_2*. Zatímco první index bude sloužit pro počítání

hesel zaslaných z druhého slovníku, počítadlo *hc_index_2* bude použito pro indexování ve slovníku prvním. Tím dosáhneme, pomocí argumentů nástroje hashcat `--limit` a `--skip`, rozdělení i prvního slovníku.

V tomto ohledu bude také vhodně implementována jistá inteligence plánovače. Můžeme očekávat, že fragmentace prvního slovníku může mít negativní dopad na výkon klienta. To například v případě, že by nám zbyl jen zlomek obsahu slovníku, který by následně dostal přidělen velmi výkonný klient. Případně, pokud by se taková situace periodicky opakovala. Počet přidělených hesel tedy bude vhodně zaokrouhlován. Pro výkonné stroje, které zvládnou v rámci úlohy vypočítat více než jedno heslo z druhého slovníku, budou plánovány úlohy bez fragmentace prvního slovníku. Implementační detaily této problematiky budou ukázány v následující kapitole.

5.7 Maskový útok

Zatímco u předchozího útoku je zřejmé, jakým způsobem je prováděno indexování ve stavovém prostoru hesel, u maskového útoku se jedná o značný problém. Nástroj hashcat totiž u maskového útoku nedokáže poskytnout reálný počet hesel a naopak pracuje se svou vlastní aritmetikou. Poskytovaný počet hesel tak záleží na délce masky a často také na obnovovaném formátu. Ve výsledku bychom mohli říct, že poskytnutý stavový prostor nemá s reálným stavovým prostorem nic společného. Bohužel tuto aritmetiku musíme využít pro plánování maskových útoků a přidělovat klientům odpovídající množství práce na základě znalosti rychlosti procesu obnovy, která je dána v heslech (respektive heších) za sekundu.

Tento problém je vyřešen naším vlastním skriptem, který byl implementován jako součást nové webové administrace. Ten dokáže na základě masky vypočítat reálný počet hesel. Při následném porovnání s hashcat stavovým prostorem zjistíme, kolik reálných hesel je reprezentováno jediným hashcat indexem. Na základě této informace pak můžeme vytvořit dílčí úlohu s odpovídající velikostí.

V samostatném nástroji hashcat máme možnost s využitím parametrů `--increment-min` a `--increment-max` specifikovat minimální a maximální délku hesla s danou maskou. Tento útok však nástroj hashcat provede jako více útoků s jednotlivými maskami. V distribuovaném prostředí jsme tuto situaci vyřešili s využitím tabulky *fc_mask*. Jediný balík nyní může obsahovat více masek a to nejen těch stejných s odlišnou délkou. Při plánování tohoto útoku jsou pak jednotlivé masky procházeny sekvenčně a postupně přidělovány jednotlivým výpočetním uzlům.

Ve výsledku tak klient počítá vždy s jedinou maskou a to ve stanoveném rozsahu indexů. Ty jsou na klientské straně zadány pomocí hashcat parametrů `--limit` a `--skip`. Pokud navíc modul Generator naplánuje dílčí úlohu, která dosahuje až na konec stavového prostoru dané masky, je vytvořena úloha pouze s daným začátkem stavového prostoru, bez omezení velikosti. Jedná se tedy o pojistku, pokud by došlo k nesrovnalosti v převodu reálných hesel a hashcat indexů. Klient s takovou úlohou prověří masku vždy až do konce. Díky tomu nemůže dojít k vynechání části masky, ve které by se například nacházelo heslo.

5.8 Hybridní útoky

Pokud se podíváme na funkčnost hybridního útoku z pohledu plánování, zjistíme, že se velmi podobá kombinačnímu útoku. Opět zde máme na jedné straně množinu hesel uloženou ve slovníku. Na straně druhé je maska, která taktéž reprezentuje jistou množinu hesel. Bohužel

zde opět narážíme na problém s reprezentací stavového prostoru nástrojem hashcat. Ten uvádí pro hybridní útoky stavový prostor rovný počtu hesel ve slovníku. V kombinačním útoku jsme měli navíc možnost zaslat v druhém slovníku pouze malý počet hesel a rozsah ověřovaných hesel v prvním slovníku kontrolovat argumentem `--limit`. To samé se bohužel nedá provést u hybridních útoků, protože nemáme prostředky, jak zaslat pouze část masky. V praxi by se tak při vyzkoušení jediného indexu stavového prostoru musela ověřit kompletní maska a to v rámci jediné dílčí úlohy. Ve výsledku by tak délka úlohy byla nekontrolovatelná a často v řádech několika dní, místo uživatelem zvoleného času.

Pro řešení tohoto problému bychom mohli aplikovat naivní postup popsaný v sekci kombinačního útoku 5.6. Tedy generovat dopředu všechny kombinace a tyto následně distribuovat jednotlivým klientům jako klasický slovníkový útok. Nicméně zde opět narážíme na zvýšení prostorové složitosti na kvadratickou, což by vedlo ke zdlouhavému přenosu všech kombinací hesel k ověření.

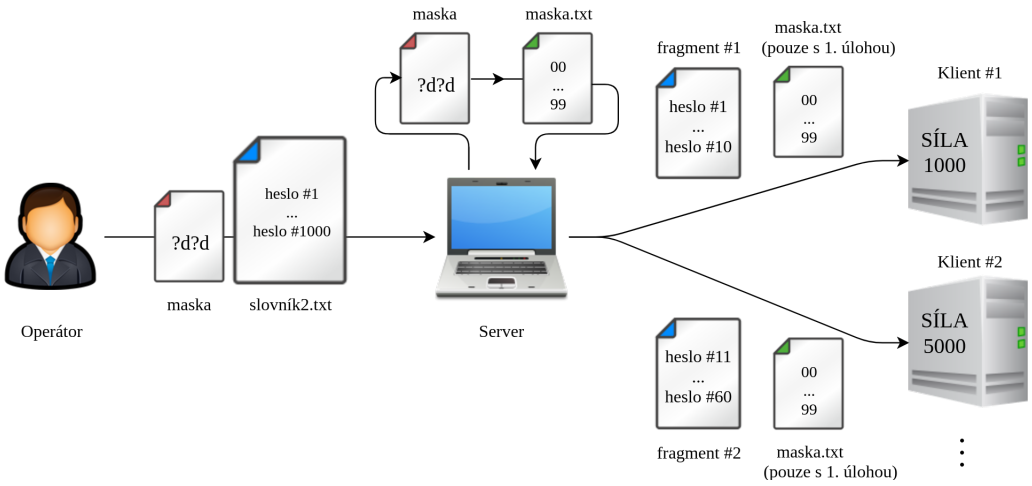
Další možností by bylo tento proces provádět až na klientské části – tedy například klientu zaslat jedno heslo s onou dlouhou maskou spolu s informací, kolik hesel má v dané úloze ověřit. Klient by si následně vygeneroval možné kombinace a indexování by prováděl až nad nimi. Tato metoda s sebou ovšem nese několik problémů. První z nich je ten, že Generator při přidělování úlohy hybridního útoku nemá k dispozici informaci o reálném ani hashcat stavovém prostoru masky, pouze samotného útoku, který se rovná, jak jsme již zmiňovaly, počtu hesel ve slovníku. Museli bychom tedy provádět vlastní propočet reálného stavového prostoru a ten přepočítávat na hashcat formát. Dalším problémem by mohla být velikost generovaného slovníku na klientské straně. U delších masek by mohlo generování trvat velmi dlouho a výsledný slovník by mohl zabírat gigabajty pevného disku. Navíc je problematické vygenerovaný slovník zachovat napříč jednotlivými úlohami, protože aktuálně na klientské straně neexistuje unikátní identifikátor jednoho obnovovaného souboru. Runner následně nemá informaci o tom, kdy by mohl vygenerovaný slovník smazat. Pokud by se generování provádělo při každé úloze znovu, mohlo by se jednat o ještě pomalejší variantu, než první navrhované naivní řešení.

Konečné řešení, které je kompromisem mezi generováním jednoho velikého slovníku a složitými procesy na klientské straně, je transformace tohoto útoku na kombinační útok. Při vytvoření hybridního útoku se ze zadané masky vygeneruje slovník hesel, pomocí hashcat nástroje *maskprocessor*. Modul Generator a stejně tak klient budou následně s tímto útokem pracovat jako s kombinačním, kde jsou na vstupu zadané dva slovníky. Diagram tohoto řešení je vidět na obrázku 5.9. Na hybridní útok lze také aplikovat Markovovské řetězce, které ovlivní postup generování hesel z masky. To bude také zmíněno v následující sekci 5.9.

5.9 Využití Markovových řetězců

Použití útoku s definovanou pravděpodobnostní maticí pro generování hesel v předem daném pořadí je možné pouze u těch typů útoků, které využívají masky. Jmenovitě je to tedy primárně útok maskový, ale také oba útoky hybridní.

V prvním zmíněném útoku bude nutné všem klientům distribuovat binární *hcstat* soubor, který právě tuto matici reprezentuje. I když je tento soubor v aktuální verzi nástroje hashcat (3.6.0) značně veliký (asi 32MB), při použití již několikrát zmíněného *sticky* příznaku je doba jeho přenosu zanedbatelná vzhledem k délce samotného útoku a to i v pomalejších sítích. Co se týče plánování útoku, nic se nemění a bude se používat stejný princip, jako u původního maskového útoku. Je však nutné zajistit, aby všichni klienti, účastnící se



Obrázek 5.9: Reálné provedení hybridních útoků – transformace na kombinační útok

výpočtu, používali totožný *hcstat* soubor. V opačném případě by totiž probíhalo generování hesel odlišně a docházelo by k přeskakování velké části hesel.

Vzhledem k tomu, že hybridní útoky budou převáděny již při jejich tvorbě na kombinační, generování hesel z masky bude probíhat již na serveru. Pro tento způsob generování můžeme využít nástroje *statsprocessor*, již zmíněného v sekci 3.4. Ten nám umožňuje kromě masky zadat na vstup i pravděpodobnostní *hcstat* soubor. Výstupem pak bude seznam hesel, vygenerovaný v daném pořadí. Vzhledem k tomu, že implementace zmíněného programu se používá i uvnitř nástroje hashcat při generování vstupních hesel pro lamač v reálném čase, mělo by být samotné generování velmi rychlé a nemělo by tak zpomalovat proces tvorby balíčku.

5.10 Použití pravidel

Slovníkový útok s využitím pravidel se velmi podobá útoku kombinačnímu. Máme zde totiž jeden slovník a k němu soubor pravidel, jejichž počet nám násobí stavový prostor původního slovníku. Nástroj hashcat udává tento prostor rovný počtu řádků ve slovníku, což je opět identické s kombinačním útokem. Zde byl stavový prostor roven počtu hesel v prvním slovníku. Přístup k provedení tohoto útoku bude tedy stejný.

Každému klientu se s první úlohou zašle celý soubor pravidel. Následně se bude zasílat takový počet hesel, který je výsledkem vzorce (*síla_stroje / počet_pravidel*).

Nevýhodou je, že zde nemůžeme použít hashcat argumenty `--limit` a `--skip` a minimální počet hesel ověřených v jedné úloze je tedy roven počtu pravidel. Jedná se o situaci, kdy bychom výpočetnímu uzlu zaslali jediné heslo ze slovníku a klient by měl za úkol aplikovat na něj všechna pravidla. To je však, v tomto případě, přijatelné. Pravidla jsou většinou tvořena ručně a generování milionů náhodných pravidel by nemělo valný smysl. I přes jejich přijatelnou velikost však bude soubor s pravidly zasílán společně s příznakem *sticky*, který zajistí, že soubor na klientské stanici setrvá po celou dobu výpočtu balíčku.

Pro představu funkcionality slovníkového útoku s pravidly je možné zhlédnout opět obrázek 5.8, reprezentující provedení kombinačního útoku. Jediná změna je zasílání souboru pravidel místo souboru „slovník1.txt“. Na klientu se pak spustí slovníkový útok s využitím zaslaných pravidel.

Kapitola 6

Implementace úprav

V následujících sekcích budou popsány implementační detaily výše navržených úprav. Primárně se bude jednat o popis struktury implementovaného modulu Generator. Zaměříme se na implementaci jednotlivých útoků navržených výše, včetně využití Markovových řetězců a pravidel. Ukážeme, jaké zajímavé problémy se při implementaci vyskytly a jaké bylo jejich řešení.

Dále zde budou shrnuty úpravy modulu Assimilator, kde jsou zpracovávány příchozí výsledky od klientů. Tento modul byl oproti původní verzi pouze upraven. Jeho hlavní částí je funkce `assimilate_handler`, která se volá z nekonečné smyčky, implementované systémem BOINC. Z toho důvodu by kompletní refaktorizace neměla valný význam, protože zde nemáme možnost vytvoření samostatného projektu, jako u modulu Generator, a celá funkcionalita by se vždy musela vyskytovat ve zmíněné funkci.

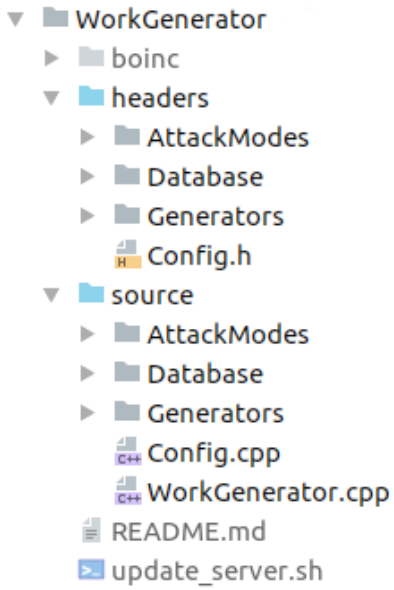
V této kapitole bude také ukázán způsob implementace skriptu XtoHashcat, jehož funkčnost jsme navrhli v sekci 5.4. Kromě toho zde popíšeme také způsoby získávání jednotlivých extrakčních skriptů a jejich případné úpravy.

6.1 Implementace modulu Generator

Nový modul Generator byl implementován v jazyce C++, kvůli kompatibilitě se systémem BOINC. Při implementaci bylo využíváno převážně vývojového prostředí CLion se studentskou licencí.

První výzvou bylo dosáhnout funkční kompilace projektu, která by zahrnovala všechny potřebné hlavičky a knihovny a vytvořila by tak funkční binární soubor. Nástroj BOINC využívá systém *automake*, ve kterém je definován překlad ukázkového modulu Generator. Vzhledem k tomu, že funkcionalita tohoto modulu přímo závisí na daném BOINC projektu, je zbytečné tvořit zcela samostatný binární soubor. Místo toho byl upraven kompilační skript nástroje BOINC tak, aby zahrnoval všechny námi vytvořené soubory. Pro tyto účely byl vytvořený krátký bash skript, `update_server.sh`, který celý proces automatizuje. Při spuštění přesune všechny zdrojové soubory na dané místo v běžícím projektu BOINC, následně provede jejich sestavení, pozastaví běžící démony, nahraje aktualizované binární soubory na správné místo a projekt restartuje. Celý proces tak zabere pouze několik sekund, místo zdlouhavých minut, kdybychom tento proces prováděli ručně.

Jak již bylo naznačeno při návrhu v sekci 5.1, projekt bude členěn do několika sekcí. Fundamentálně je rozdělen do složek *headers*, kde se nachází všechny hlavičkové soubory a složky *source*, kde se nachází všechny zdrojové soubory. Hlavní funkce `main` je imple-



Obrázek 6.1: Ukázka adresářové struktury projektu WorkGenerator

mentována v souboru `WorkGenerator.cpp` a má pouze několik řádků. Kromě zpracování BOINC argumentů, které bylo z velké části převzato z původní implementace, obsahuje pouze vytvoření instance třídy *CSimpleGenerator* a zavolání její nekonečné smyčky. Dále je projekt rozčleněn do 3 sekcí:

- **Generators** – obsahuje třídy, zodpovědné za logiku distribuce dílčích úloh,
- **Database** – obsahuje třídy pro reprezentace položek databáze a manipulace s nimi,
- **AttackModes** – obsahuje implementaci jednotlivých navržených útoků.

Kromě toho ještě projekt obsahuje soubory `Config.cpp`, respektive `Config.h`. V těch jsou definovány globální proměnné, jako jsou například názvy tabulek databáze, cesty ke slovníkům, pravidlům a *hcstat* souborům na serveru. Dále je zde definován jmenný prostor `Tools`, ve kterém se aktuálně vyskytují pouze funkce pro ladící výstup. Obecně se tedy jedná o proměnné a funkce, které mají být přístupné odkudkoli z projektu.

Popsaná adresářová struktura projektu je vidět na obrázku 6.1.

6.1.1 Sekce Generators

Tato sekce projektu obsahuje třídy, zajišťující logiku generování jednotlivých útoků pro jednotlivé klienty. Nachází se zde abstraktní třída *CAbstractGenerator*, která by měla sloužit jako předek pro všechny implementované třídy typu Generator. Aktuálně definuje rozhraní ve formě funkce `run`, kterou musí implementovat všichni potomci. Dále tato abstraktní třída definuje funkci `calculateSecondsIcdf2c`, která obsahuje algoritmus pro adaptivní úpravu času plánovaného útoku v závislosti na stavu výpočtu, kterou mohou využít všichni potomci [9]. Stejně tak funkci `activateJobs`, která zajišťuje synchronizaci s ostatními BOINC démony.

Aktuálně je implementován jediný potomek typu Generator a to je *CSimpleGenerator*. Jak již bylo zmíněno ve fázi návrhu, viz 5.1.1, tento generátor bude fungovat velmi přímočaře. Pro každý běžící balíček se pokusí vytvořit dílčí úlohu, a to pro každého přiřazeného

klienta. Nově připojeným uzlům generuje úlohu typu *bench_all*, viz 5.3. Klientům, kteří se nově účastní lámání specifického souboru, také zadá úlohu *benchmark*, pro opětovné změření výkonu na daném formátu. Detaily implementace těchto útoků budou popsány níže.

Dále je schopen ukončovat běh výpočtu z několika důvodů, jako je pozastavení uživatelem, vypršení stanoveného časového limitu, vyčerpání stavového prostoru nebo samozřejmě z důvodu nalezení hesla. Co se týče generování samotných lámacích dílčích úloh, udržují se v systému vždy dvě, jedna je zaslána klientu a druhá je připravena pro budoucí komunikaci, aby se předešlo zbytečnému zdržování při generování.

Kompletní funkčnost nekonečné smyčky *CSimpleGenerator* je lépe vidět v algoritmu 6.2.

Znovupřidělování nedokončených úloh

Jak již bylo naznačeno ve zmíněném pseudokódu, *CSimpleGenerator* podporuje také generování nových útoků z nedokončených úloh, označených příznakem *retry*. Toto značení vznikne buď zpracováním chybného výsledku modulem Assimilator nebo vypršením přidělené doby dané úlohy. Ke stanovení této doby je využita funkčnost systému BOINC. Ten dokáže stanovit dobu, do které je očekáván výsledek zaslané dílčí úlohy. Ta lze v databázi našeho systému nastavit v položce *default_job_timeout_factor* tabulky *fc_settings* jako násobek zadané doby výpočtu dílčí úlohy. Po vypršení této doby je nastaven sloupec *server_state* BOINC tabulky *result* na hodnotu 5, do stavu *Over*. Tato hodnota je nicméně nastavena i v případě úspěšného dokončení úlohy. Databázová spoušť, která tuto změnu sleduje, tedy kromě nastavení této hodnoty také ověřuje, zda je aktuální časové razítko větší nebo rovno hodnotě *report_deadline*, kam systém BOINC ukládá právě časový limit pro dokončení úlohy. Pokud jsou všechny podmínky splněny, je nastaven příznak *retry* dané položky *fc_job*. Modul Generator z této úlohy v následujícím cyklu generuje další práci.

Vytváření úlohy z již existující nedokončené úlohy pak nahrazuje část `generateJob` jednotlivých tříd *AttackMode*, viz níže. Pokud při vytváření útoku pro daného klienta již existuje úloha, zkopírovaná z nepovedeného útoku, daná třída toto detekuje a již se nepokouší o generování nové úlohy a úpravu aktuálních indexů daného balíčku. Kopírování nedokončených úloh probíhá vždy přednostně před generováním nových úloh. Pokud je navíc balíček ve stavu *Finishing*, nedochází ke generování nových úloh vůbec. Pokud v tomto případě již neexistuje úloha typu *retry*, ukončí klient práci přechodem do stavu *Done*.

Z důvodu chystané implementace pokročilejší třídy Generator nebylo navrženo a implementováno žádné inteligentní dělení nedokončené úlohy do menších podúloh. Daná úloha je tedy vždy zkopírována do nové v poměru 1 : 1.

Zastavení výpočtu

Tato sekce se týká problematiky 5.1.5, kde byly probírány možnosti zastavení, restartu, případně pozastavení a pokračovaní ve výpočtu. V případě, kdy chce správce projektu výpočet pozastavit, uvede se daný balíček do stavu *Finishing*. V tom případě se již negenerují nové úlohy a čeká se na dokončení těch probíhajících. Do takového stavu se balíček dostane i po vyčerpání stavového prostoru nebo překročení stanoveného časového limitu.

Po dokončení všech prováděných úloh pak může nastat jedna z následujících možností.

- **Vyčerpání stavového prostoru** – Aktuální index je roven stavovému prostoru, stav balíčku je tedy nastaven na *Exhausted*.

```
1: while true do
2:     Smaž klienty z fc_host, jejichž balíček je dokončen – buď úspěšně (Finished) nebo
       neúspěšně (Exhausted)
3:     Pokud nějaký balíček přesáhl dobu ukončení, nastav jeho status na Finishing
4:     for Pro všechny běžící balíčky (status >= 10) do
5:         Pokud balíček ještě nemá nastavený čas startu, nastav ho na aktuální
6:         Pokud má k sobě balíček nějaké vázané masky, ulož je k balíčku
7:         Umísti do tabulky fc_host klienty, kteří mají zájem účastnit se výpočtu

           {Benchmark}
8:         for Pro každého přiřazeného klienta ve stavu 0 (Benchmark) do
9:             if Klient ještě nemá žádný benchmark naplánovaný then
10:                Naplánuj benchmark
11:            end if
12:        end for

           {Výpočet}
13:        for Pro každého přiřazeného klienta ve stavu 1 (Normal) do
14:            if Počet naplánovaných úloh pro klienta je >= 2 then
15:                Pokračuj na dalšího klienta
16:            end if
17:            if Stav balíčku == 10 (Running) then
18:                Vygeneruj novou úlohu podle typu útoku, případně z retry
19:            else if Stav balíčku je 12 (Finishing) then
20:                Vygeneruj novou úlohu z retry. Pokud taková neexistuje, nastav kli-
                   enta do stavu 3 (Done)
21:            end if
22:        end for

           {Kontrola stavu}
23:        if Stav balíčku je 12 (Finishing) and Nejsou vygenerovány žádné úlohy then
24:            if Aktuální čas > Čas ukončení then
25:                Nastav balíček do stavu 4 (Timeout)
26:            else if Aktuální index >= Maximální index then
27:                Nastav balíček do stavu 2 (Exhausted) – stavový prostor vyčerpán
28:            else
29:                Nastav balíček do stavu 0 (Ready) – balíček pozastaven
30:            end if
31:        end if
32:    end for
33:    Čekej na synchronizaci ostatních BOINC démonů
34: end while
```

Obrázek 6.2: Ukázka funkčnosti modulu SimpleGenerator

- **Překročení časového limitu** – Aktuální čas je větší než stanovený *time_end*, stav balíčku je nastaven na *Timeout*.
- **Pozastavení** – Pokud neplatí ani jedna z předchozích podmínek, byl balíček pozastaven a jeho stav je nastaven na *Ready*.

Balíček může být restartován pouze pokud neběží, tedy není ve stavu *Running* ani *Finishing*. To znamená, že aktuálně není vykonáván žádný výpočet na klientské straně. Řešení využívající aktivní komunikace s klienty pro přerušení prováděných výpočtů a tedy možnost okamžitého zastavení výpočtu ze strany serveru nebyla implementována. Pokud tato vlastnost bude v budoucnu vyžadována, může být součástí nově vznikajícího pokročilého modulu Generator.

6.1.2 Sekce Database

V modulu Generator je nutné pracovat s daty uloženými v MySQL databázi, jejíž schéma jsme popsali výše. Pokud bychom při každém přístupu k těmto datům využívali SQL dotaz, proces generování by byl velmi pomalý. Kvůli tomu jsou položky jednotlivých tabulek reprezentovány třídami. Při potřebě práce s položkou databáze je načten celý záznam do objektu. V projektu definujeme následující třídy:

- **CHost** – reprezentuje záznamy tabulky *fc_host*,
- **CJob** – reprezentuje záznamy tabulky *fc_job*,
- **CMask** – reprezentuje záznamy tabulky *fc_mask*,
- **CPackage** – reprezentuje záznamy tabulky *fc_package*.

Právě dynamické vytváření těchto typů objektů způsobovalo úniky paměti v předchozí implementaci modulu Generator. Jak bylo zmíněné výše v sekci 5.1.1, pro řešení tohoto problému využijeme chytré ukazatele, které dokáží automaticky uvolňovat paměť objektů, na které již neexistuje reference. To je zajištěno tak, že všechny zmíněné třídy mají privátní konstruktor. Jediný způsob pro jejich tvorbu je statická třída `create`, která vrací právě chytrý ukazatel *shared_pointer* na nově vzniklý objekt. Díky tomuto rozhraní není možné vytvořit novou instanci bez jejího následného uvolnění.

Práce s databází je zapouzdřena ve třídě *CSqlLoader*. Ta pak definuje vlastní rozhraní, kterými můžou ostatní moduly v projektu s databází komunikovat. Definice samotných SQL dotazů a jejich provádění se tedy vyskytuje pouze v této třídě. Ta je instancována ve třídě *CSampleGenerator* a následně předávána všem modulům, které ji potřebují.

Samotné položky z databáze, které vyhovují určitému kritériu, jsou vraceny ve vektoru z funkce `load`. Jedná se o funkci, která má typ určený šablonou a vrací vektor chytrých ukazatelů na zvolený typ databázového objektu. Použití šablon nám umožní implementovat pouze jedinou funkci pro získávání všech typů objektů z databáze.

Co se týče vzniku samotného objektu z informací v databázi, do konstruktoru jednotlivých databázových tříd se předá mapa, v kódu definovaná jako `DbMap`. Ta obsahuje seznam dvojic `<nazev_sloupce : hodnota_sloupce>`. Díky tomuto zpracování je funkčnost kódu nezávislá na existenci pro tento modul nezajímavých sloupců, případně dovoluje přidávat nové sloupce do databáze, bez nutnosti upravovat způsob zpracování, což byl ve starém prototypu modulu Generator problém. Proměnná `hodnota_sloupce` je vždy typu řetězec. Ve

starém modulu byla použita knihovna *boost* s funkcí `lexical_cast` pro převod řetězce na správný, většinou číselný formát. Této závislosti jsme se v novém modulu Generator zbavili, protože pro převod na číselné typy si vystačíme se standardní knihovnou. Tedy s využitím funkcí `stoul`, `stoull`, apod.

Jedinou výjimku pro vytváření databázových objektů tvoří třída *CJob*, kterou je možné vytvářet také pomocí ručně zadaných hodnot z kódu, a to kvůli vytváření nových dílčích úloh. Takto vytvořený objekt pak slouží jako šablona pro vznik nové položky v databázi. Naopak načítání objektů *CJob* z databáze slouží pro kopírování nových úloh z úloh s příznakem *retry*. Toho můžeme využít a již v konstruktoru nastavovat jisté údaje typické pro zkopírovanou úlohu, jako je nastavení příznaku *duplicated* nebo vynulování příznaku *retry* ze zkopírovaného záznamu.

6.1.3 Sekce AttackModes

Poslední sekcí projektu je složka *AttackModes*. Ta obsahuje definice tříd pro jednotlivé typy útoků. Všechny pak mají společného předka *AttackMode*. Jedná se o abstraktní třídu, která definuje rozhraní pro vytváření jednotlivých útoků. Tímto rozhraním je především virtuální funkce `makeJob`, která nejprve vytvoří novou úlohu typu *CJob*, případně použije již existující úlohu, která jí byla předána pomocí funkce `setJob`. Následně provede úkony nutné pro vytvoření pracovní jednotky, kterou zašle klientu. Kromě toho tato abstraktní třída uchovává členské proměnné, jako jsou chytré ukazatele na balíček a klienta, ke kterým se daná úloha váže, nebo ukazatel na objekt *CSqlLoader*.

Způsob implementace tohoto rozhraní je u každého útoku značně odlišný. Celkem definujeme 6 tříd útoků:

- **CAttackBench** – třída pro vytváření úloh typu *benchmark*,
- **CAttackDict** – třída pro vytváření klasických slovníkových útoků,
- **CAttackRules** – třída pro slovníkové útoky s využitím souboru pravidel,
- **CAttackCombinator** – třída pro vytváření kombinačních a hybridních útoků,
- **CAttackMask** – třída pro vytváření klasických maskových útoků,
- **CAttackMarkov** – třída pro maskové útoky s využitím *hcstat* souboru.

V následujících odstavcích se podíváme na implementační detaily jednotlivých útoků.

Úloha benchmark

Vytváření úloh typu *benchmark* je implementováno ve třídě *CAttackBench*. I když se nejedná přímo o metodu útoku, z hlediska systému BOINC je to úloha jako každá jiná a princip jejího vygenerování je tedy shodný s útoky.

Při vytváření této úlohy je možné předat parametr *duration* libovolný, protože doba úlohy typu *benchmark* se nestanovuje – z principu chceme, aby proběhlo měření co nejrychleji. Dále proběhne generování nové položky *CJob*, která má kromě identifikace balíčku a klienta ostatní parametry, jako je velikost stavového prostoru, počáteční index nebo ID masky, nulové.

Následně je vytvořen konfigurační soubor z databázového sloupce *config* daného balíčku a je k němu připojen záznam o typu útoku:

```
|||mode|String|1|b|||
```

Jedinou výjimku tvoří generování úlohy pro balíček s ID rovným hodnotě 1. Tento balíček je vyhrazen pro úlohy typu *bench_all* a místo módu `b` je vložen mód `a`. Tak klient pozná, že má provést kompletní benchmark místo klasického. O korektní zpracování odpovědi na tuto úlohu se pak postará modul Assimilator.

Nakonec je pomocí BOINC rozhraní vytvořena úloha, která je určena výhradně danému klientu a záznam vygenerovaného objektu *CJob* je zkopírován do tabulky *fc_job*. Tato fáze je u všech níže popisovaných útoků stejná, pouze je toto rozhraní voláno s různými vstupními soubory, které jsou určeny šablonou, korespondující s daným útokem.

Slovníkový útok

Třída *CAttackDict* definuje klasický slovníkový útok. Generování nové úlohy je velmi přímočaré – hodnota aktuálního výkonu daného klienta se vynásobí ideální dobou trvání útoku, který je dodán již zmiňovanou funkcí `calculateSecondsIcdf2c`. Tím získáme počet hesel, který chceme v dané dílčí úloze vyzkoušet.

Po aktualizaci hodnoty *current_index* daného balíčku, která slouží pro plánování následující úlohy, můžeme přejít k vytváření vstupních souborů pro klienta. Nejprve je načten konfigurační soubor z databáze a doplněn o následující řádek:

```
|||mode|String|1|n|||
```

Tato úprava říká, že se jedná o klasickou výpočetní úlohu, nikoliv o úlohu typu *benchmark*. Tento řádek je do konfiguračního souboru přidán i ve všech níže popsaných útocích. Dále je vytvořen soubor *data*, který obsahuje samotný vstup do nástroje hashcat – tedy například heš, jehož původní podobu hledáme. Tento soubor je nutné vytvořit u všech lámacích úloh a zaslat jej klientu.

Nyní již zbývá pouze poslední soubor a tím je fragment slovníku, jehož obsah má klient ověřit. Ten je vytvořen následovně. Po otevření vstupního slovníku balíčku je nejprve přeskočen počet řádků, rovnající se hodnotě *start_index* dané úlohy, a to s využitím standardní funkce `getline`. Přeskočená hesla již byla použita v předchozím plánování a tedy přidělena v jiných úlohách. Následně je pomocí stejné funkce získán počet hesel, rovný hodnotě *hc_keyspace* dané úlohy. Jednotlivé čtená hesla jsou ukládána do nového souboru *dict1*, který slouží jako vstupní fragment slovníku pro klienta. Po přečtení dostatečného počtu hesel zavřeme oba soubory a všechna vstupní data tak máme připravena. Je vhodné poznamenat, že v průběhu čtení vstupního slovníku je kontrolován příznak `eof`. Pokud je nastaven, znamená to, že jsme narazili na konec slovníku. Balíček je v takovém případě přepnut do stavu *Finishing*, kde se čeká na dokončení všech prováděných úloh.

Nakonec se opět vytvoří úloha v systému BOINC, nový záznam ve Fitcrack tabulce *fc_job* a úloha je zaslána klientovi.

Útok s využitím pravidel

Z pohledu nástroje hashcat se tento útok liší od slovníkového útoku pouze zadáním jednoho parametru. Z hlediska plánování však musíme počítat se zvětšeným stavového prostoru hesel, i když v terminologii nástroje hashcat se hodnota *hc_keyspace* nezměnila.

Ve funkci `generateJob` tedy postupujeme stejně, jako u předchozího útoku – vytvoříme objekt *CJob*, jehož stavový prostor *hc_keyspace* je rovný počtu reálných hesel, které dokáže klient za daný čas ověřit. Nicméně zatím neupravujeme hodnotu *current_index* daného balíčku. Následně spočteme reálný počet zasílaných hesel, který získáme následujícím vztahem: $(hc\_keyspace/rules\_size)$, kde *rules_size* je počet pravidel. Získaná hodnota udává počet hesel, zasílaných ve fragmentu slovníku. Touto hodnotou aktualizujeme *hc_keyspace* dané úlohy a také ji přičteme k hodnotě *current_index* daného balíčku, pro plánování další úlohy.

Fragment slovníku následně vznikne stejným způsobem jako u slovníkového útoku. Hodnota vstupní souboru *config* je upravena opět přidáním módu `n` a ze vstupního heše je vytvořen soubor *data*. Soubor s pravidly je zaslán celý, nicméně v šabloně útoku má nastaven příznak *sticky*. Tím je zajištěno, že pokud má daný soubor stejný název a obsah, je zaslán pouze při první úloze daného balíčku.

BOINC úloha je pak generována stejným způsobem, jak bylo zmíněno výše.

Kombinační útok

Jak již bylo zmíněno v sekci návrhu, 5.10, útoky kombinační a útoky s pravidly se z hlediska plánování velmi podobají. Nejprve tedy proběhne generování nového objektu *CJob* zcela totožným způsobem. Stejně tak úprava souboru *config* i tvoření souboru *data* jsou identické s předchozím útokem.

Zatímco však v útoku s pravidly posíláme kompletně soubor pravidel a fragmentujeme první slovník, zde je situace opačná. První slovník je posílán celý, s příznakem *sticky*, zatímco slovník druhý se fragmentuje. Situace je zde navíc o to složitější, že předpokládáme i případy, kdy nemůžeme ověřit kompletní kombinaci $(n \times 1)$ hesel. Implementace plánování se tak rozdělí do 3 větví.

- **Klient má dostatečný výkon** – Jedná se o situaci, kdy má klient dostatečný výkon pro ověření alespoň $(n \times 1)$ hesel a zároveň ještě nedošlo k fragmentování prvního slovníku pomocí parametrů `--skip` a `--limit`. V tom případě je uzlu zaslán vždy celý první slovník, pokud ho ještě nevlastní, a k hesel z druhého slovníku. Klient má pak za úkol vyzkoušet všechny kombinace $(n \times k)$ hesel.

- **První slovník je již fragmentovaný** – Pokud jsme již začali fragmentovat první slovník, pomocí `--skip` a `--limit`, je to vyznačeno nenulovou hodnotou *start_index_2*, která se používá právě pro tyto účely. V tomto případě musíme pokračovat ve fragmentaci. Pokud má klient dostatečný výkon, dokáže dokončit všechny kombinace i. hesla v druhém slovníku. Nastaví tak *start_index_2* opět do nuly a posune plánování balíčku na hodnotu $(i + 1)$. V opačném případě se aktuální index balíčku neupravuje a klient pouze vyzkouší další část hesel z prvního slovníku v kombinaci s jedním hesel z druhého slovníku. Tato úloha se vyznačuje tím, že se v druhém slovníku vždy zasílá pouze jediné heslo a v souboru *config* je uložena informace o tom, v jakém rozsahu indexů prvního slovníku se pohybujeme.

- **Klient nemá dostatečný výkon** – V případě, že klient nemá dostatečný výkon na vyzkoušení alespoň $\frac{(n \times 1)}{2}$ kombinací hesel, musí zahájit fragmentování prvního slovníku. Pokud tedy uzel dokáže ověřit alespoň polovinu hesel z prvního slovníku, uvažujeme, že má dostatečný výkon. Díky tomu předejdeme situacím, kdy by klient ozkoušel většinu hesel z prvního slovníku a následující klient by pak dostal úlohu

s velmi malým stavovým prostorem, která by představovala velkou režii, oproti efektivnímu výpočtu. Tento útok se vyznačuje stejnými parametry, jako předchozí odrážka, pouze má vždy nastaven parametr `--skip` na hodnotu nula. Hodnota *start_index_2* je pak nastavena na počet zaslaných hesel v prvním slovníku.

Maskový útok

Platí, že maskový útok, definovaný třídou *CAttackMask*, je nejefektivnější z hlediska přenosu dat. Naopak však patří, spolu s kombinačním útokem, k těm nekomplexnějším z hlediska plánování. Ve funkci `generateJob` musíme nejprve najít masku, ze které bude útok vytvořen. Každý balíček se může vázat k více maskám, hledáme tedy takovou, která ještě nebyla kompletně ověřena. To znamená, že její hodnota *current_index* je menší, než hodnota *hc_keyspace*. Následně vypočítáme počet *hc_indexes*, který dokáže daný klient za daný čas spočítat. Tuto hodnotu přičteme jednak k plánovací hodnotě balíčku *current_index*, ale také ke stejnojmenné hodnotě vybrané masky. Zároveň při tvorbě objektu *CJob* poznačíme do databáze ID masky, ze které byla úloha generována. To pak můžeme využít při kopírování nedokončené úlohy s příznakem *retry*. V opačném případě bychom totiž nevěděli, jakou masku daná úloha využila.

Následně opět vytvoříme vstupní soubory *config* a *data*. Do konfiguračního souboru navíc doplníme textovou podobu masky, která má být při útoku využita, a parametry `start_index` a `hc_keyspace`, jejichž hodnoty jsou na klientu použity jako argumenty `--skip` a `--limit`. Pokud navíc chceme dokončit generování hesel z dané masky až do konce jejího stavového prostoru, parametr `hc_keyspace` vynecháváme. Tím dosáhneme toho, že klient přeskočí daný počet hesel a ověří celý zbytek masky. Pokud by tedy došlo na serveru k chybě výpočtu *hc_keyspace*, na klientu nikdy nemůže dojít k vynechání hesel.

Útok s využitím Markovových řetězců

Plánování tohoto útoku je shodné s útokem maskovým. Využití *hcstat* souboru nám nijak nemodifikuje stavový prostor *hc_keyspace* – s jedinou výjimkou a tou je využití parametru `--markov-threshold`, který omezí hloubku prohledávání pravděpodobnostní matice, viz 3.2. Pokud je ale stavový prostor nastaven správně do databáze při vytváření útoku, plánování to nijak neovlivní.

Důvod, proč je tento útok implementován separátní třídou *CAttackMarkov*, je využití jiné šablony vstupních souborů. S tím také souvisí jiná práce s rozhraním BOINC. To by mělo za důsledek velké množství rozhodovacích větví a tedy špatnou čitelnost programu.

Jediným rozdílem je tedy navíc zasílání *hcstat* souboru, ve kterém je definováno pořadí generování hesel. Tento soubor je, podobně jako například soubor s pravidly, zasílán pouze při první dílčí úloze daného balíčku. Toho je dosaženo, jak již bylo zmíněno, příznakem *sticky* v šabloně útoku. Vzhledem k velikosti tohoto souboru, 32MB, je to značná optimalizace a zmenšení režie přenosu vstupních dat.

6.2 Implementace modulu Assimilator

Modul Assimilator slouží pro zpracování příchozích výsledků výpočtu dílčích úloh. Jak bylo diskutováno v kapitole návrhu 5.2, byly provedeny úpravy, které zajišťovaly schopnost tohoto modulu zpracovat výsledky všech tří typů popsaných útoků.

Po ověření, zda se jedná o validní výsledek a zjištění, od kterého klienta přišel a ke kterému balíčku se výsledek vztahuje, dojde k větvení podle zmíněného typu úlohy. Čtení všech popsaných hodnot uložených v souboru výsledku probíhá standardní funkcí `fscanf` s odpovídajícím typem. Následně je ověřeno, zda došlo k úspěšnému čtení a získaná hodnotou je uložena. V původním modulu docházelo vždy ke čtení řetězce, který byl bez kontroly vkládán do příkazu SQL. Potenciální útočník tak mohl zasláním nevalidního výsledku způsobit pád modulu Assimilator, v horším případě i provést útok SQL Injection. To již díky provedeným úpravám nehrozí.

V případě, že dojde k chybě čtení nebo je zaslán kód různý od nuly výsledku typu *benchmark*, je naplánován nový, a to s exponenciálně rostoucím časovým odstupem. Tato technika se často používá i v komunikačním protokolu BOINC a zajišťuje tak velkou škálovatelnost systému – díky zvětšujícímu se odstupu nezahltí špatně konfigurované stanice server nesmyslnými požadavky. Pokud dojde ke stejné chybě ve výpočtu, je daná lámací úloha označena příznakem *retry* a klient, na kterém k chybě došlo, je uveden do stavu 0 (*Benchmark*) a musí znovu provést měření svého výkonu.

Pokud je naopak úloha typu *benchmark* ukončena úspěšně, dojde k aktualizace nově přidané tabulky *fc_benchmark*. V té je pak uložen poslední známý výkon daného formátu na daném stroji. V případě klasické úlohy typu *benchmark* dojde k aktualizaci pouze jedné položky, odpovídající aktuálnímu formátu. V případě úlohy *bench_all* dojde k přidání, respektive aktualizaci všech úspěšně změřených formátů – tedy těch, jejichž hodnota je větší než nula.

Jedním ze základních rozdílů mezi starým a novým modulem Assimilator je pak manipulace s výsledkem po jeho zpracování. Zatímco v původním modulu docházelo k jeho mazání, nově je výsledek pouze označen příznakem *finished*. Díky tomu máme k dispozici historii prováděných úloh, kterou pak můžeme zpracovat vhodným způsobem a vizualizovat ve webové administraci.

Další zajímavou úpravou je přepočet získaného výkonu u *benchmark* úlohy patřící k balíčku typu maskového útoku. Zatímco měření vždy vrací počet reálných hesel za sekundu, plánování maskového útoku probíhá za pomocí hashcat indexů. Proto před uložením aktuálního výkonu výpočetního uzlu proběhne vydělení hodnoty poměrem (*keyspace*/*hc_keyspace*).

Funkčnost zpracování jednoho výsledku modulem Assimilator je znázorněno algoritmem 6.3.

6.3 Implementace XtoHashcat

Skript pro automatickou extrakci vstupních dat, XtoHashcat, byl implementován v jazyce Python 3 a to podle objektového návrhu popsaného v sekci 5.4. Jeho spouštění tedy probíhá následovně.

```
./XtoHashcat.py <Cesta_k_souboru> [-f <Mód>]
```

Kromě již zmíněných tříd v návrhu obsahuje implementovaný skript dodatečnou statickou třídu, nazvanou *StaticHelper*. V té se nachází jednak seznam podporovaných formátů a také metoda pro analýzu získaného heše. Ta se volá z vybrané instance třídy *Format*, která o sobě následně získané informace vytiskne.

Nástroj XtoHashcat aktuálně podporuje extrakci vstupních dat z následujících formátů:

- **Balík MS Office** – dokumenty formátu MS Word, Excel a PowerPoint, od verzí 97 do současnosti,

```
1: Zjisti zdroj výsledku – klient a balíček
2: if Typ výsledku == Benchmark then
3:     if Výsledek je v pořádku (kód 0) then
4:         if Mód útoku == Mask then
5:             Proveď přepočet na hashcat-indexy
6:         end if
7:         Ulož výkon klienta a čas benchmarku do databáze
8:     else
9:         Naplánuj nový benchmark za exponenciálně se zvyšující počet minut
10:    end if
11: else if Typ výsledku == Normal then
12:    if Heslo nalezeno (kód 0) then
13:        Nastav balíček do stavu 1 (Finished)
14:        Zašli všem pracujícím klientům pokyn k ukončení výpočtu
15:        Označ probíhající dílčí úlohy jako dokončené
16:        Ulož do databáze čas výpočtu, a nalezené heslo
17:        Vlož výsledek do fc_hashcache, pokud tam není
18:    else if Heslo nenalezeno (kód 1) then
19:        if Úloha měla dostatečnou velikost then
20:            Uprav výkon klienta podle délky výpočtu poslední úlohy
21:        end if
22:        Ulož do databáze čas výpočtu a uprav počet ověřených hesel
23:    else
24:        Zruš všechny prováděné úlohy daným klientem
25:        V databázi těmto úlohám nastav příznak retry
26:        Nastav výkon klienta na 0, stav na 0 (Benchamrk)
27:        Ukončí vykonávání této funkce, aby úloha nebyla označena za dokončenou
28:    end if
29: else if Typ výsledku == Bench_all then
30:    if Výsledek je v pořádku (kód 0) then
31:        Ulož čas benchmarku do databáze
32:        for Pro všechny naměřené formáty do
33:            Ulož do databáze naměřenou rychlost nebo aktualizuj existující záznam
34:        end for
35:    else
36:        print  Důvod chybného výpočtu
37:    end if
38: end if
39: Nastav úloze příznak finished
```

Obrázek 6.3: Ukázka funkčnosti modulu Assimilator

- **PDF** – všechny podporované verze dokumentů PDF nástrojem hashcat,
- **RAR** – všechny podporované verze archivů RAR5 nástrojem hashcat,
- **ZIP** – všechny podporované verze archivů WinZIP nástrojem hashcat,
- **7-Zip** – všechny verze archivů 7-Zip.

Informace z těchto formátů jsou extrahovány s využitím následujících open-source extrakčních skriptů a programů:

- ***office2hashcat*** – skript v jazyce Python, dostupný v repozitáři systému, Hashstack[1],
- ***7z2hashcat*** – skript v jazyce Perl, dostupný v repozitáři systému Hashstack,
- ***pdf2hashcat*** – skript v jazyce Python, dostupný v repozitáři systému Hashstack,
- ***rar2john*** – binární soubor, který je součástí nástroje John the Ripper[2],
- ***zip2john*** – binární soubor, který je součástí nástroje John the Ripper.

Všechny nástroje byly otestovány na všech dostupných verzích jednotlivých formátů. U skriptu *office2hashcat* byly provedeny menší úpravy, protože docházelo ke špatné detekci u verzí Office Excel 2003, což bylo způsobeno naivním hledáním podřetězce, který se u nově vzniklých dokumentů nenachází. Tento přístup byl modifikován na spolehlivější detekci [19].

Dále nástroje *rar2john* a *zip2john*, jak již napovídá název, neposkytují vstup do nástroje hashcat, ale do nástroje John the Ripper. Nicméně ořezáním několika znaků na začátku a konci tohoto výstupu můžeme vždy bezpečně získat správný formát, čehož je také využito.

Funkcionalita nástroje XtoHashcat je shrnuta v algoritmu 6.4.

```
1: Zpracuj vstupní argumenty
2: if Zadán argument -f then
3:     Nastav formát ručně
4: else
5:     Zjisti formát na základě signatury a koncovky
6: end if
7: if Nebyl nastaven validní formát then
8:     Ukonči vykonávání skriptu
9: end if
10: Zavolej odpovídající skript pro extrakci heše
11: if Formát == ZIP or Formát == RAR then
12:     Převeď z formátu john do formátu hashcat
13: end if
14: Proveď analýzu získaného heše
15: print  Získaný vstup do nástroje hashcat
16: print  Detekovaný hashcat formát
```

Obrázek 6.4: Ukázka funkčnosti skriptu XtoHashcat

[1] https://github.com/stricture/hashstack-server-plugin-hashcat
[2] https://github.com/magnumripper/JohnTheRipper

Kapitola 7

Experimenty

V této kapitole budou provedeny experimenty s nově implementovaným nástrojem Fitcrack. Měření bude probíhat primárně ve výzkumné laboratoři C304, kde je umístěno celkem dvacet osobních počítačů s následujícími parametry:

- CPU – Intel® Core™ i5-3570K,
- GPU – NVIDIA GTX 1050 Ti,
- paměť – DDR3 8 GB,
- základní deska – Intel DB75EN.

Pro naše účely je důležitá hlavně moderní grafická karta, která je základem pro běh nástroje hashcat. Na počítačích jsou přítomny operační systémy jak Windows, tak Linux. Vzhledem k lepší optimalizaci ovladačů GPU a také aktuálnější verzi našeho nástroje Runner bude pro experimenty využíván primárně systém Windows, na který byly nainstalovány jak grafické ovladače, tak nástroje BOINC Client a BOINC Manager.

V první části experimentů bude provedeno měření výkonu našeho systému a jeho porovnání s předchozí verzí. To bude demonstrováno nejprve srovnáním výkonu na straně výpočetních uzlů. Ukážeme zde, jakého zrychlení jsme dosáhli díky využití nástroje hashcat na klientské straně. Právě toto zrychlení, spolu s podporou nových formátů, je hlavním důvodem této práce.

Následně také porovnáme rychlosti generování starého a nového modulu Generator. I když tento výkon nemá na výslednou rychlost lámání valný vliv, toto porovnání nám může zodpovědět několik otázek. Nový modul musí oproti starému provádět mnohem více operací a je také zatížen objektovým návrhem, využitím chytrých ukazatelů apod. Na druhou stranu došlo k menší optimalizaci z hlediska volání SQL dotazů. Uvidíme tedy, zda, a jaký vliv měly jednotlivé provedené úpravy. Uvidíme také, jak rychle probíhá generování pro jednoho klienta a jaké je zpomalení pro rostoucí počet klientů. Tento graf nám dokáže zodpovědět, zda je náš systém použitelný i ve větším měřítku, například pro stovky připojených klientů.

V druhé části experimentů se pak zaměříme na praktické předvedení funkčnosti jednotlivých navržených útoků v distribuovaném prostředí laboratoře C304. Nakonec provedeme také experimenty, ve kterých bude výpočet pozastaven, nebo jichž se účastní nespolehlivé výpočetní uzly, které se v průběhu výpočtu odpojují. Tímto způsobem prokážeme schopnost našeho systému fungovat spolehlivě i v nestabilním prostředí.

7.1 Porovnání rychlostí na straně klienta

Pro srovnání rychlostí původního lámacího nástroje s nástrojem hashcat jsme použili nejnovější verzi samostatného nástroje Fitcrack a verzi 3.6.0 nástroje hashcat. V průběhu experimentů vyšla již nová verze nástroje hashcat, 4.1.0, nicméně při vypracování této práce byla využívána vždy zmíněná verze 3.6.0. Novější verze bude pravděpodobně vyžadovat menší úpravy jak na straně serveru, tak klientů. Pro experimenty byly vybrány formáty, které podporují oba zmíněné nástroje. Následuje seznam zvolených formátů:

- Dokumenty MS Office verze 97–2016,
- Archivy WinZIP, RAR3, RAR5 a 7-Zip,
- Dokumenty PDF revize 4 a 5.

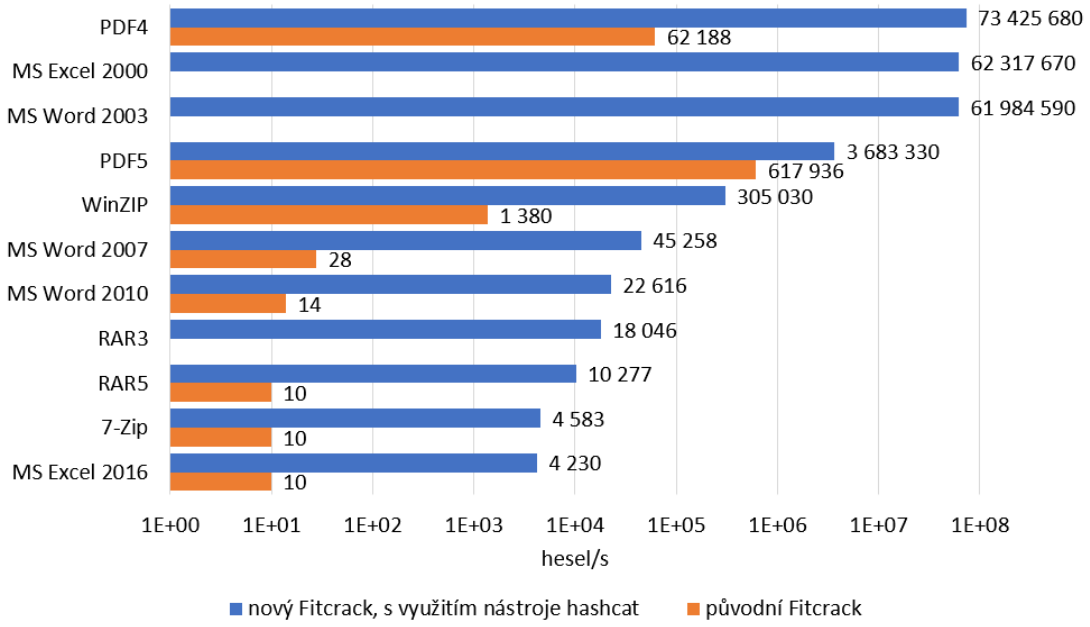
Rychlost nástroje hashcat byla získána s využitím vestavěné funkce benchmark, která byla u každého formátu spuštěna celkem desetkrát a výsledná rychlost byla vypočítána jako aritmetický průměr získaných hodnot. U nástroje Fitcrack bylo využito taktéž vestavěné funkce benchmark a to o délce šedesáti sekund. Výsledná hodnota je pak dána průměrnou rychlostí za zvolenou dobu.

Při pokusu o kompilaci původního nástroje Fitcrack se ukázalo, že pro systém Windows nebyla kompilace nikdy zprovozněna. Proto musely být experimenty provedeny na systému Linux. Bohužel by instalace nejnovějších grafických ovladačů na zastaralý systém Linux v laboratoři C304 vyžadovala velké zásahy. Z toho důvodu jsme zvolili pro tyto experimenty zcela jiný počítač, který již měl grafické ovladače nainstalované. Jednalo se o osobní počítač střední třídy s následujícími parametry:

- CPU – AMD FX-8320 3,5 GHz,
- GPU – Gigabyte R9 280X,
- paměť – DDR3 16 GB,
- operační systém – Ubuntu 14.04 LTS.

Kromě toho vyžadoval nástroj Fitcrack pro kompilaci množství knihoven v zastaralých verzích, které již nejsou normálně dostupné. Konečným řešením tedy bylo použití existujícího binárního souboru, který byl vytvořen pro využití v poslední verzi distribuovaného prototypu Fitcrack.

Některé údajně podporované formáty v původním nástroji Fitcrack navíc vůbec nefungovaly. Jednalo se o formáty MS Office verzí 2000 a 2003, které hlásily neexistující *Cracker*. To bylo pravděpodobně způsobeno špatnou konverzí z nástroje Wrathion do nástroje Fitcrack. Dále nástroj Fitcrack končil s chybou *segmentatiopn fault* na formátu RAR3. Další problémy se pak objevily u novějších formátů, u kterých bylo nutné ručně omezovat velikosti *global work size* (GWS). Pokud byla zvolená hodnota příliš velká, došlo k pádu grafického ovladače a nástroj vykazoval nesmyslně velkou rychlost, i když výpočet vůbec neprobíhal. Naopak při volbě dostatečně nízké hodnoty GWS u náročných formátů naměřil benchmark rychlosti v řádů několika hesel za sekundu. Tato rychlost je pravděpodobně několikrát větší při obnově delších úloh, nicméně zjišťování nejlepších laboratorních podmínek pro běh tohoto nástroje není součástí této práce. Pro měření byla vždy použita hodnota získaná z poskytovaného benchmarku.



Obrázek 7.1: Porovnání rychlosti nástroje hashcat a původního samostatného nástroje Fitcrack

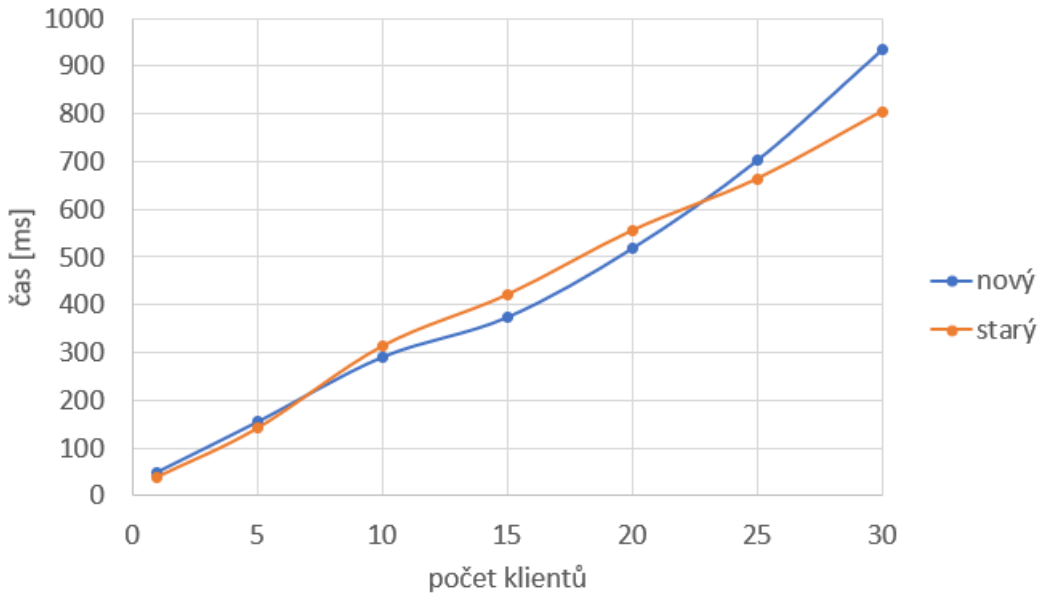
Ani nástroj hashcat však nefungoval bez problémů. U formátů RAR3 a RAR5 se objevila chyba, poukazující na nedostatečnou paměť GPU, tedy podobný problém jako s GWS výše. Nástroj hashcat na to nicméně upozornil a po přidání argumentu `-n 64`, který tuto hodnotu omezoval, fungovalo vše bez problémů.

Přes všechny popsané problémy proběhlo měření a porovnání hodnot, které můžeme vidět na obrázku 7.1. Osa Y představuje jednotlivé formáty, zatímco na ose X jsou rychlosti ověřených hesel za sekundu v logaritmickém měřítku. Vzhledem k obrovskému zrychlení, které je místy více než tisícinásobné, můžeme tedy konstatovat, že nahrazením klientské části nástrojem hashcat jsme z prototypu, fungujícího jen v laboratorních podmínkách, vytvořili prakticky použitelné řešení.

7.2 Porovnání rychlosti generování

V další části měření se zaměříme na rychlost generování práce na straně serveru. Jak již byl zmíněno v úvodu této kapitoly, při volbě dostatečné délky výpočtu dílčí úlohy je režie pro vygenerování úlohy zanedbatelná. Systém BOINC je navíc navržen pro masové využití napříč celým světem, kde se jednoho projektu můžou účastnit miliony dobrovolníků. To v případě našeho nástroje Fitcrack není reálné, protože využití necílí na dobrovolnické výpočty, ale spíše na *grid computing*.

Na druhou stranu rychlost generování a primárně také strategie plánování jednotlivých útoků budou hlavním rozdílem mezi naším řešením a konkurenčními nástroji, zmíněnými v sekci 4.3, které ná klientské straně využívají rovněž nástroje hashcat. Pokud by například tyto nástroje transformovaly kombinační útok na slovníkový, případně by jej řešily jen pomocí argumentů `--skip` a `--limit`, režie, respektive čas jedné dílčí úlohy, by velmi narostly. Z důvodu stále aktivního vývoje jak na klientském nástroji Runner, tak na webové administraci však nebude v současné době možné se zmíněnou konkurencí provést srovnání.



Obrázek 7.2: Porovnání rychlostí nového modulu Generator a jeho staršího prototypu

Všechny zmíněné komponenty jsou totiž součástí celku a mají vliv nejen na rychlost obnovy, ale také na použitelnost celého řešení.

Pro srovnání rychlosti generování byly použity dva moduly Generator. Jeden z nich je nově vyvinuté řešení, které bylo popisováno výše v sekci 6.1 a využívá modulární architektury a množství funkcí ze standardu C++11, které zvyšují čitelnost kódu. Druhým modulem Generator bude prototyp tohoto programu, který byl využíván v první části akademického roku pro testovací účely. Ten je implementován v jednom zdrojovém kódu – je tedy velmi nečitelný a má další nedostatky, které jsme blíže rozebrali v sekci 5.1.

Měření proběhlo na soukromém notebooku s procesorem Intel® Core™ i5-7300HQ, kde byl lokálně nainstalován BOINC server spolu s MySQL databází. Byla měřena vždy jedna iterace generování úlohy pro různý počet klientů, od jednoho až po třicet současně připojených výpočetních uzlů. Pro každou variantu bylo provedeno 50 iterací. Mezi každou iterací se modul Generator pozastaví a čeká na synchronizaci ostatních BOINC démonů. Tato doba uspání nebyla měřena, protože je ovlivněna velkým množstvím okolních faktorů. Čas uspání je navíc většinou o několik řádů delší, než samotné generování a ve výsledku bychom tak neviděli téměř žádný rozdíl.

Jako typ útoku byl vybrán maskový útok, protože má ze všech výše popsaných útoků nejmenší režii pro vytváření vstupních souborů. Nedochází zde například ke kopírování slovníků, jako u slovníkového útoku. Tím eliminujeme nezajímavou režii otevírání a čtení souborů apod.

Před samotnými experimenty byly také odstraněny všechny ladící výpisy, které by mohly mít vliv na měřený čas. Samotné měření času pak bylo uskutečněno pomocí standardní knihovny *chrono*.

Na obrázku 7.2 již vidíme získané hodnoty. Na tomto grafu jsou vyneseny průměrné časy jedné iterace modulu Generator s různým počtem klientů zapojených do výpočtu. Vidíme, že jejich rychlosti jsou velmi podobné, tedy tak, jak jsme předpokládali. Starší modul Generator při vetším počtu klientů generuje práci o několik milisekund rychleji,

což je pravděpodobně způsobeno jednoduchou strukturou daného programu, která však způsobovala jeho špatnou rozšiřitelnost. Zmíněné urychlení však není výraznou výhodou – oproti době klasického útoku v rámci několika hodin nebo dní nemá několik milisekund vliv na výslednou dobu výpočtu.

Navíc zde vidíme, že čas, potřebný pro vygenerování nových pracovních jednotek, roste lineárně. Důležitým závěrem tohoto experimentu je tak skutečnost, že i na středně výkonném notebooku je možné obsloužit stovky či tisíce současně lámajících uzlů. I pro takový počet připojených klientů totiž proběhne generování pracovních jednotek v rámci několika málo sekund. Jediným omezením tak může být nedostatečná šířka komunikačního kanálu, což ale při využití výpočetních uzlů na lokální síti většinou nenastane.

V příloze D se nachází kompletní naměřená data ve formě grafů. Z nich jde mimo jiné vyčíst, že nový modul Generator je při menším počtu připojených klientů rychlejší a průměrnou rychlost pak snižují občasné výkyvy.

7.3 Experimenty s jednotlivými útoky

V této kapitole demonstrujeme funkčnost jednotlivých navržených a implementovaných útoků. Distribuovaný výpočet bude prováděn v laboratoři C304 a to většinou s využitím deseti výpočetních uzlů, jejichž parametry byly popsány v úvodu této kapitoly. Server bude umístěn na samostatném stroji a s klienty bude komunikovat skrz síť Internet. Útoky budou navržené tak, aby v přiměřeném čase vždy úspěšně skončily nalezením hesla. Délka jedné dílčí úlohy bude stanovena na 60 sekund. To je v praxi velmi neefektivní, protože režie nutná pro komunikaci v takovém případě tvoří výraznou část celého výpočtu. Pro demonstrační účely jednotlivých útoků to však nevadí. Dočkáme se pouze menšího, než lineárního zrychlení.

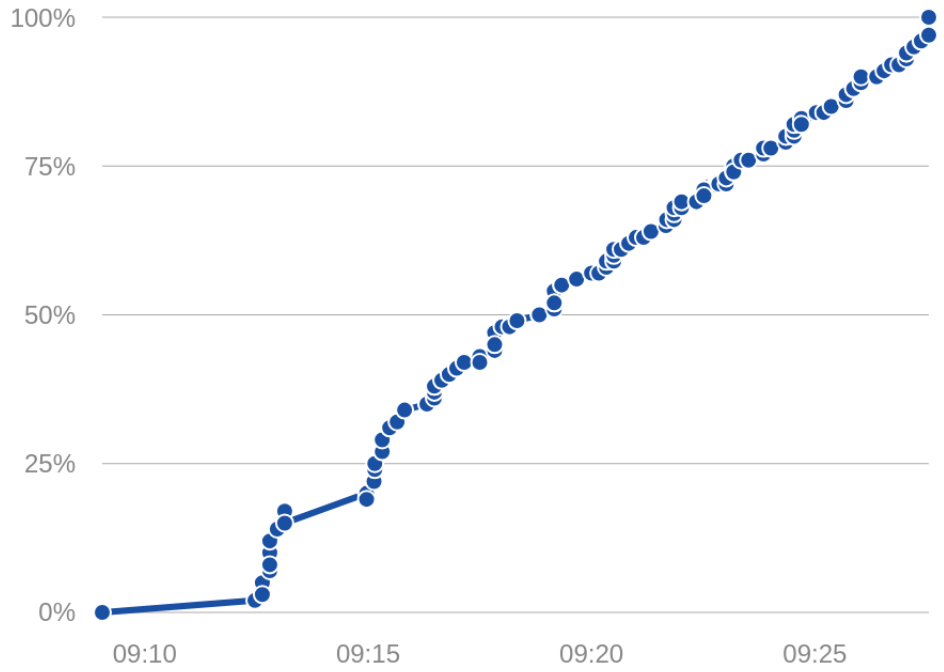
Čas každého distribuovaného útoku pak srovnáme s čistým časem výpočtu, čímž určíme zrychlení oproti sekvenčnímu lámání. U všech výpočtů bude nástroj hashcat spouštěn s parametrem `-w 3`, který zajistí běh s dostatečným výkonem. Jedná se o druhý nejvyšší výkonnostní stupeň – ten nejvyšší stupeň je totiž experimentální a může docházet k problémům se zamrzáním grafického ovladače.

Výsledky budou zobrazeny na několika grafech, které jsou převzaty z aktuálně vyvíjeného webového rozhraní nástroje Fitcrack. Tyto grafy zobrazují jednak velikosti jednotlivých dílčích úloh, dále pak průběh lámání nebo podíl výpočetních uzlů na výpočtu. Kromě slovního popisu tedy i uvidíme, jak výpočet v praxi probíhal.

7.3.1 Slovníkový útok

Potenciálně nejméně efektivním útokem v distribuovaném prostředí je útok slovníkový. Zde totiž musíme každému klientu distribuovat všechny kandidáty k ověření. V naši předchozí práci jsme ukázali, že tato efektivita vzrůstá s rostoucí složitostí obnovovaného formátu [12]. Toho zde využijeme a jako formát pro experimenty zvolíme hešovací funkci bcrypt [16]. Ta byla navržena s ohledem na rostoucí výkon počítačů a její výpočet je tak uměle zpomalený, aby se zabránilo rychlému prolomení hesla. Vestavěný benchmark nástroje hashcat změřil odhadovanou rychlost obnovy formátu bcrypt na jednom našem výpočetním uzlu s výsledkem přibližně 4 tisíce hešů za sekundu. To je velmi výrazné zpomalení například oproti SHA-1, u které lze ověřit za jedinou sekundu přibližně 2,2 miliardy hešů.

Jako seznam hesel k ověření byl zvolen známý slovník hesel *rockyou.txt*, který obsahuje asi patnáct milionů unikátních hesel. Námi zahešované heslo jsme umístili na konec tohoto



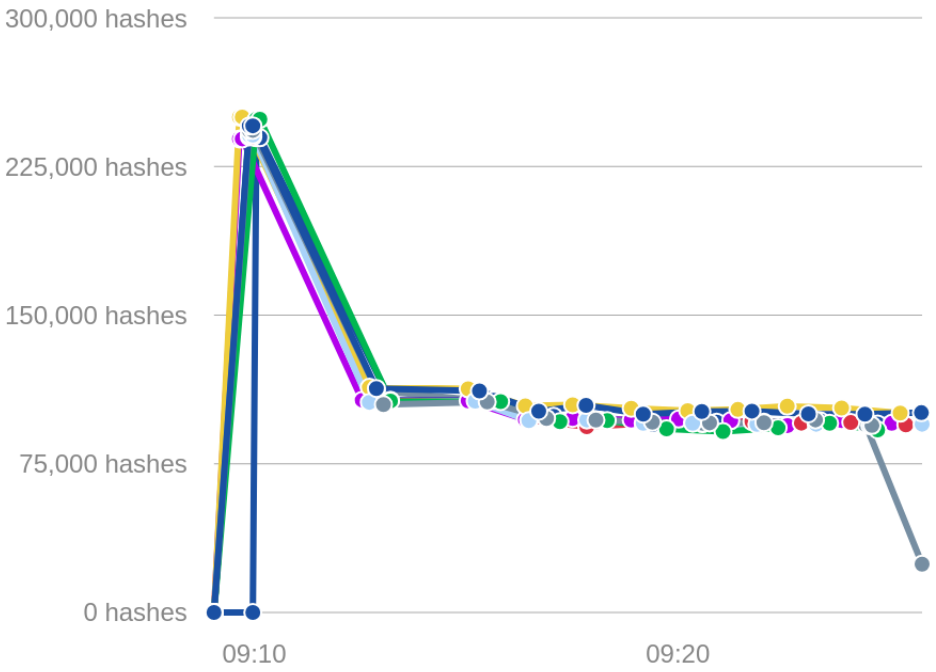
Obrázek 7.3: Průběh slovníkového útoku v čase

slovníku. Dle odhadovaného výkonu zvolené množiny výpočetních uzlů by mělo k ověření celého slovníku a tedy odhalení hesla dojít do několika minut.

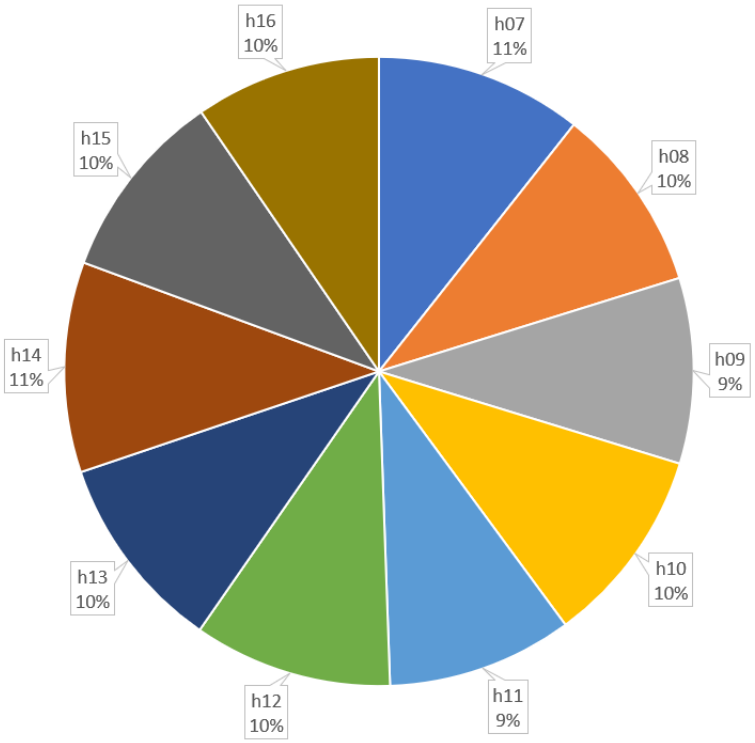
Na obrázku 7.3 vidíme průběh tohoto útoku v čase. Každý bod v grafu znamená dokončení jedné dílčí úlohy a tedy posun v prohledávaném stavovém prostoru. První dvě úlohy byly přibližně dvakrát delší než očekávaná doba 60 sekund. To bylo způsoben použitím hešovací funkce bcrypt se 6 iteracemi, jejíž výpočet trvá déle než standardní 4 iterace, které měří nástroj hashcat. To stejné je vidět také na grafu 7.4, který zobrazuje velikosti jednotlivých dílčích úloh v počtu ověřovaných kandidátů. Každý klient je zde zobrazen jinou barvou. Je však zřejmé, že po těchto dvou úlohách došlo k úpravě velikosti, která se ustálila asi na sto tisících hesel v jednotlivých zasílaných fragmentech. Zajímavá je poslední úloha, která měla asi čtvrtinovou velikost. Jednalo se o poslední fragment slovníku *rockyou.txt*, tedy zbytek, ve kterém se nacházelo hledané heslo.

Výpočet pak probíhal rovnoměrně na všech uzlech až do 100%, protože hledané heslo jsme záměrně umístili až na konec samotného slovníku. Rovnoměrné rozdělení práce je vidět také na grafu 7.5. Ten zobrazuje dělbu práce v kompletním úspěšném slovníkovém útoku mezi laboratorními výpočetními uzly *h07* – *h16*.

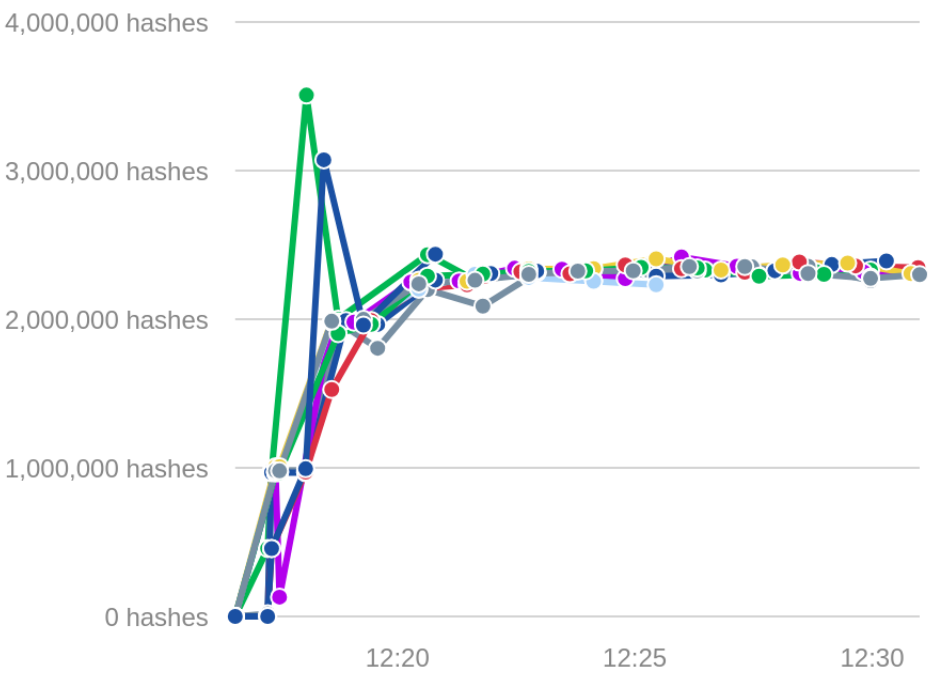
Celý slovníkový útok trval celkem 27 minut, včetně měření výkonu výpočetních uzlů. Za tuto dobu se provedlo v součtu celkem 133 minut čistého výpočtu. Vidíme zde tedy přibližně pětinásobné zrychlení oproti lámání na jednom stroji. I když dle očekávání se mělo jednat až o desetinásobné zrychlení, důvodem pro menší číslo je jednak režie při distribuci slovníkových fragmentů, ale hlavně malá doba jednotlivých dílčích úloh. Pokud počítáme, že zdržení při standardní komunikaci BOINC je deset sekund při odeslání i přijetí výsledku, celkový čas jedné šedesátisekundové úlohy se tak může zvětšit o desítky procent. Při experimentu se slovníkovým útokem s pravidly níže, v podsekci 7.3.4, uvidíme, že i při malém navýšení doby dílčích úloh dosáhneme výrazně lepších výsledků.



Obrázek 7.4: Velikost dílčích úloh slovníkového útoku v čase, klienti různými barvami



Obrázek 7.5: Podíl práce mezi jednotlivými klienty při slovníkovém útoku



Obrázek 7.6: Velikost dílčích úloh maskového útoku v čase, klienti různými barvami

7.3.2 Maskový útok

Pro demonstraci maskového útoku jsme zvolili aktuálně velmi populární funkci SHA-256. Heslo se skládalo z 9 malých písmen latinky, která navíc byla zvolena v čitelné podobě, nicméně nenesla žádný význam. Toho využijeme při srovnání s útokem Markovovými řetězci níže. Tento útok byl koncipován tak, že jsme postupně zkoušeli masky pro jednoznaková až desetiznaková hesla, celkem tedy deset masek formátu `?l` až `?l?l?l?l?l?l?l?l?l?l`.

Ze začátku docházelo k přidělení kompletního stavového prostoru masky. Ten je totiž u krátkých masek velmi malý – například 26 kandidátů pro masku `?l`, 676 kandidátů pro masku `?l?l` atd. Od masky s délkou 8 dochází již k fragmentování stavového prostoru masky. Tento jev je dobře vidět na obrázku 7.6. Zatímco u slovníkového útoku byl stavový prostor stejný pro všechny dílčí úlohy již od počátku, zde jsou na začátku výpočtu výrazné skoky. Tato velikost se následně ustálí při začátku ověřování masky s délkou osm, respektive devět. Kvůli tomu také můžeme na tomto grafu sledovat, že úlohy jsou dokončovány postupně. U předchozího útoku, který byl prezentován na obrázku 7.4, jsme mohli sledovat ze začátku spíše nárazové dokončení dílčích úloh, protože všichni připojení klienti měli stejný výkon a od počátku dostávali přiděleno podobné množství práce.

Graf rozdělení práce mezi jednotlivé uzly, který jsme pro slovníkový útok viděli na obrázku 7.5, se kvůli popsanému jevu na začátku výpočtu lišil – někteří klienti dostali přiděleno více práce, než jiní. V průběhu následujících minut se však tento rozdíl postupně smazal a ve výsledku je tak získaný graf identický.

Distribuovaný maskový útok trval celkem 15 minut. V tomto intervalu vykonali klienti 85 minut čistého výpočtu. Zrychlení je zde tedy asi 5,6-násobné. To je o něco více, než u slovníkového útoku. Nebylo totiž nutné přenášet velké fragmenty hesel. K ideálnímu lineárnímu zrychlení jsme ovšem stále daleko, kvůli velmi malé době dílčí úlohy.

Obrázek 7.7: Velikost dílčích úloh kombinačního útoku v čase, klienti různými barvami

7.3.3 Kombinační útok

Kombinační útok je z hlediska režie efektivnější, než útok slovníkový. Po počátečním přenesení prvního slovníku, který obsahuje N hesel, způsobí každé další přenesené heslo vygenerování N kandidátů na klientské straně.

Pro demonstraci tohoto útoku byl zvolen slovník hesel získaný phishingovým útokem na službu MySpace[1] v roce 2006. Tento slovník obsahuje asi 37 tisíc hesel, které budeme kombinovat se slovníkem 1000 nejpoužívanějších hesel[2]. Celkově tedy bude útok reprezentovat asi 37 milionů unikátních hesel. To je asi dvojnásobek velikosti útoku slovníkového výše. Řádově jsou si však velmi podobné a k úspěšnému nalezení hesla by tedy mělo dojít do několika minut.

Jako formát tedy opět zvolíme hešovací funkci bcrypt. Levou část hesla umístíme na konec slovníku MySpace, jako pravou část pak zvolíme poslední heslo slovníku 1000 nejpoužívanějších hesel.

Graf 7.7 zobrazuje velikosti jednotlivých dílčích úloh v čase. Na začátku opět vidíme skok, který je ovšem o něco menší, než v útoku slovníkovém. To je způsobeno volbou pouze 5 iterací ve funkci bcrypt, z důvodu většího stavového prostoru kombinačního útoku. Zajímavý je také zbytek grafu – velikosti úloh se nám rozprostřely pouze do dvou úrovní. To je způsobeno zaokrouhlováním velikosti fragmentu druhého slovníku, jak bylo zmíněno v sekcích 5.6 a 6.1.3. V praxi se tak s každou úlohou posílalo vždy 4 nebo 5 hesel. Ve zmíněném grafu vidíme, že velikosti úloh tedy odpovídají čtyřnásobku, respektive pětinásobku původních 37 tisíc hesel v prvním slovníku.

Stejně jako u předchozích útoků bylo rozložení práce mezi jednotlivé výpočetní uzly rovnoměrné. Graf průběhu byl totožný s grafem 7.3 slovníkového útoku – tedy se rovnoměrně

[1] https://news.netcraft.com/archives/2006/10/27/myspace_accounts_compromised_by_phishers.html
[2] https://github.com/danielmiessler/SecLists/blob/master/Passwords/darkweb2017-top1000.txt

zvyšoval až do 100%, protože části hledaného hesla byly záměrně uloženy na konec obou slovníků.

Distribuovaný kombinační útok trval přibližně 38 minut. V jeho průběhu vykonaly všechny výpočetní uzly celkem 240 minut užitečné práce. Vidíme zde přibližně 6,3-násobné zrychlení oproti sekvenčnímu výpočtu. Lepších výsledků oproti předchozím útokům bylo dosaženo díky delší době výpočtu – počáteční měření a zasílání velkých slovníků na začátku lámání tak již nemělo velký vliv na celkovou dobu výpočtu.

7.3.4 Slovníkový útok s využitím pravidel

V našem demonstračním slovníkovém útoku s pravidly jsme využili známého slovníku uniklých hesel phpBB[3]. Ten obsahuje přes 184 tisíc unikátních hesel. Na tento slovník jsme aplikovali soubor *best64.rule*, což je soubor se 77 pravidly, které upravují hesla nejrůznějšími, často používanými způsoby. Mezi ty řadíme například nahrazování písmen „o“, „e“, respektive „l“ za číslice „0“, „3“ resp. „1“, dále připojování číslic nebo nejpoužívanějších znaků nakonec hesla apod. Tento soubor pravidel je poskytován nástrojem hashcat.

Pokud vynásobíme stavové prostory slovníku a pravidel, získáme číslo podobné velikosti slovníku *rockyou.txt*, použitém ve slovníkovém útoku výše. Opět tedy zvolíme náročnou hešovací funkci bcrypt.

V tomto útoku, jako jediném z této množiny experimentů, jsme zvolili větší čas dílčí úlohy a to 140 sekund. Celkový útok trval 16 minut. Za tu dobu provedly výpočetní uzly 140 minut užitečné práce. I při stále velmi krátké době dílčí úlohy jsme tedy s deseti počítači dosáhli 8,75-násobného zrychlení, což se již blíží lineárnímu zrychlení, kterého jsme se snažili dosáhnout. Při předchozích experimentech s nástrojem BOINC [9] bylo ukázáno, že s prodlužující se dobou jednotlivých dílčích úloh je režie zcela zanedbatelná a zrychlení tedy lineární.

7.3.5 Maskový útok s využitím Markovových řetězců

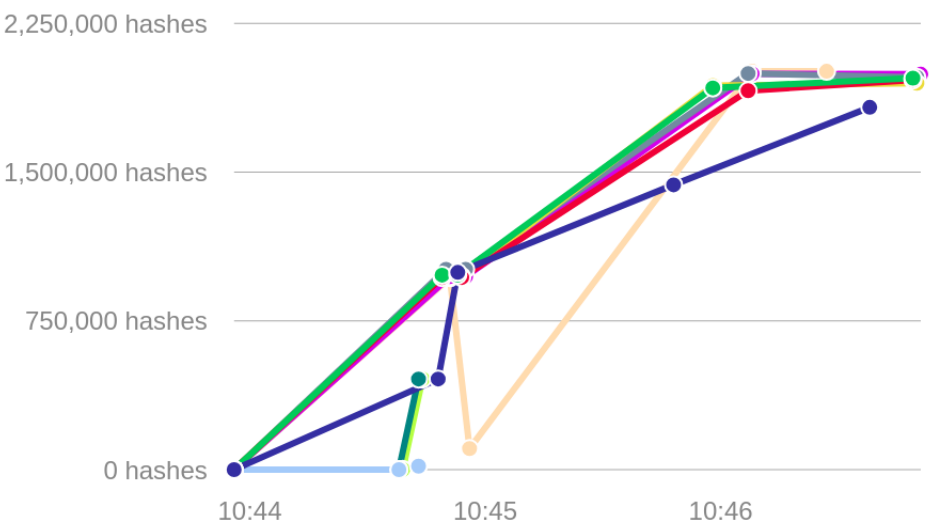
Při popisu implementace tohoto typu útoku v sekci 6.1.3 jsme ukázali, že tento útok je shodný s útokem maskovým, pouze s dodatečným parametrem, určujícím pořadí generovaných hesel. Kromě samotné funkčnosti útoku s využitím Markovových řetězců tento experiment ukazuje, že s pomocí tohoto přístupu můžeme heslo získat dříve, než klasickým maskovým útokem. A to za předpokladu, že hledané heslo není zcela náhodné, ale je tvořeno existujícími slovy, případně frázemi, které jsou pro člověka jednoduše zapamatovatelné.

Příkladem takového hesla je fráze `aerotaner`, která byla použita jako vstup hešovací funkce i u maskového útoku výše. Nyní provedeme opět maskový útok s deseti maskami. Budeme však navíc distribuovat i binární *hcstat* soubor, který vznikl s pomocí již několikrát zmíněného slovníku *rockyou.txt*.

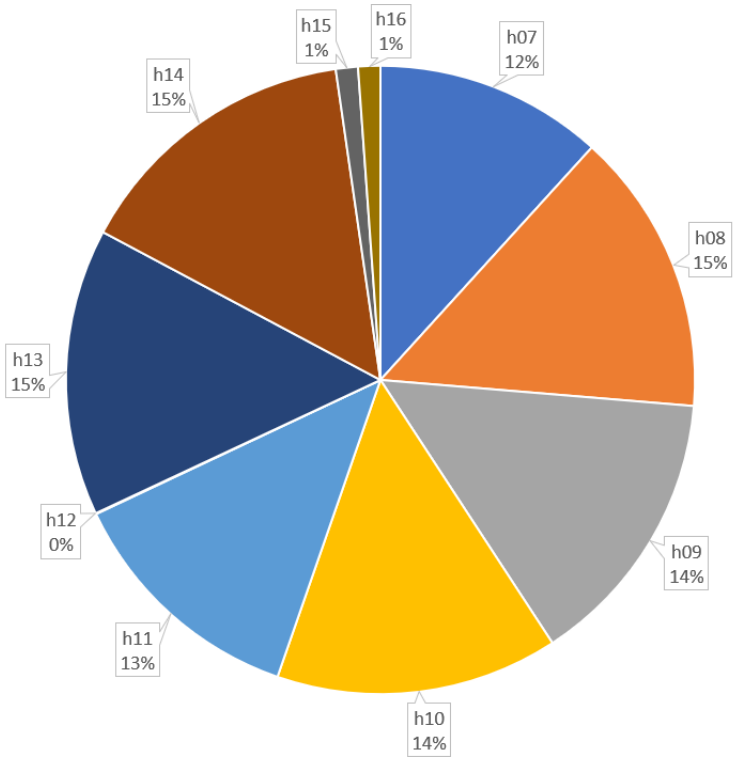
Na grafu 7.8 vidíme, že celý útok byl úspěšně ukončen do 3 minut a podobá se začátku grafu 7.6. To je výrazné urychlení oproti původnímu maskovému útoku bez *hcstat* souboru, který trval přibližně 15 minut. Důvodem tohoto zrychlení je vygenerování námi zvoleného hesla hned na začátku stavového prostoru 9-znakových hesel.

Celková užitečná práce na všech uzlech se však rovná 6 minutám. Tento experiment je tak příkladem velmi neefektivního útoku v distribuovaném prostředí. Jednotlivým klientům byly generovány úlohy pro velmi malé masky a režie tak byla mnohonásobně větší, než samotné ověření. Zatímco jeden stroj by všechny masky do velikosti sedm prověřil v rámci

[3] https://wiki.skullsecurity.org/Passwords



Obrázek 7.8: Velikost dílčích úloh maskového útoku v čase s využitím *hcstat* souboru, klienti různými barvami



Obrázek 7.9: Podíl práce mezi jednotlivými klienty při útoku s využitím *hcstat* souboru

několika sekund, v systému BOINC celá procedura distribuce a zpracování výsledku trvala desítky sekund. To mělo také vliv na rozdělení práce mezi uzly, které je vidět v grafu 7.9. Zde vidíme, že některé uzly ověřily právě miniaturní masku a k další práci se již nedostaly.

7.4 Experimenty s pozastavením výpočtu a nespolehlivými klienty

Cílem poslední části experimentů bude demonstrovat funkčnost přerozdělování nedokončených úloh a to buď z důvodu selhání klienta nebo jeho odpojení. Dále také demonstrujeme chování systému při pozastavení výpočtu. Společným rysem těchto úloh bude volba dlouhé masky se zmíněnou náročnou hešovací funkcí bcrypt. Cílem těchto měření tak nebude nalezení hesla, ale ukázka chování našeho systému v různých situacích.

7.4.1 Přidělování nedokončených úloh

Prvním experiment ukazuje chování našeho systému při odpojení některých výpočetních uzlů. Tohoto výpočtu se pro lepší orientaci ve výsledném grafu účastnili pouze 4 klienti. Asi po 15 minutách, při prohledání přibližně 25% celého stavového prostoru, byly pozastaveny tři ze čtyř účastnících se uzlů. To vidíme na grafu 7.10, kdy se přibližně v 13:50 přestaly pro tyto uzly generovat úlohy. Dále si všimněme, že v následujících minutách zelený uzel zopakoval poslední vygenerované úlohy odpojených klientů, na které server nedostal odpověď. Navíc ve 14:00 dojde k vypršení stanoveného limitu, odpojení klienti jsou uvedeni do stavu *benchmark* a je jim přidělena úloha typu *benchmark*. Nicméně tito klienti zůstávají pozastaveni a tedy nereagují.

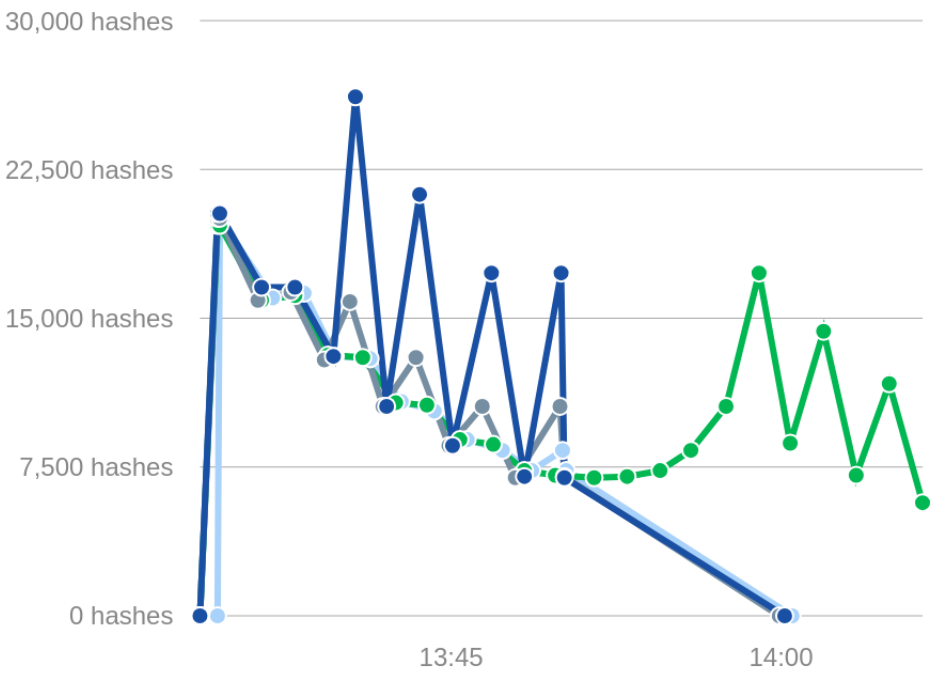
Tuto situaci je možné také sledovat na grafu 7.11. Všimněme si, že po zastavení 3 uzlů se postup ve výpočtu zpomalil přibližně na jednu čtvrtinu. Na tomto experimentu vidíme, že nástroj Fitcrack dokáže přerozdělovat nedokončené úlohy klientů, kteří neočekávaně ukončili výpočet. Díky tomu nemůže dojít k vynechání části stavového prostoru hesel, ve kterém by se mohlo nacházet hledané heslo, respektive vstupní hodnota kryptografického heše.

7.4.2 Pozastavení výpočtu

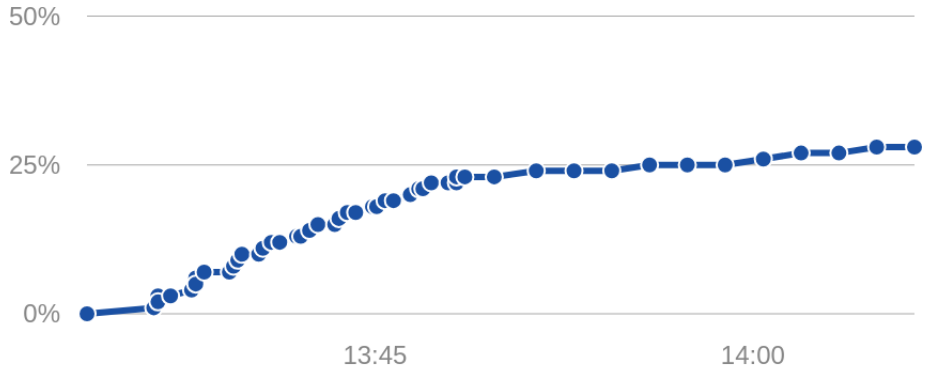
V posledním experimentu ukazujeme funkčnost pozastavení výpočtu distribuovaného systému Fitcrack. Jak bylo popsáno v sekci 5.1.5, pozastavením se balíček uvede do stavu *Finishing* a přestanou se generovat nové úlohy. Následně jsou dokončeny všechny naplánované dílčí úlohy a balíček je uveden do stavu *Ready*.

Toto chování je vidět na následujících grafech. Experiment proběhl tak, že přibližně po deseti minutách byl výpočet pozastaven. Na obrázku 7.12 vidíme, že poslední úloha byla vygenerována v čase 13:01, kdy v následujících sekundách došlo k pozastavení výpočtu. Na grafu 7.13 je naopak vidět, že výpočet pokračoval a poslední výsledek byl doručen v čase 13:04, tedy přibližně po dvojnásobku stanovené doby výpočtu jedné dílčí úlohy.

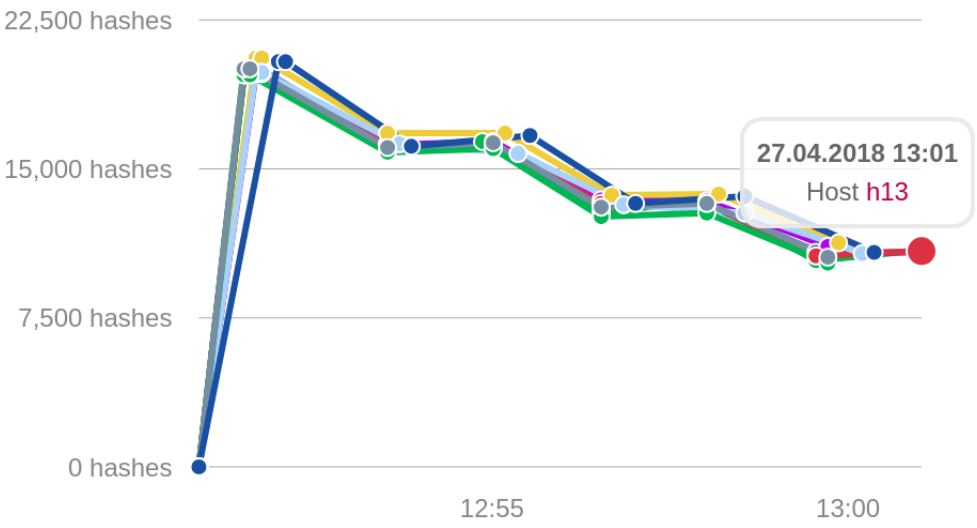
Na tomto experimentu vidíme, že funkčnost pozastavení výpočtu se v praxi chová podle očekávání, tedy tak, jak bylo navrženo.



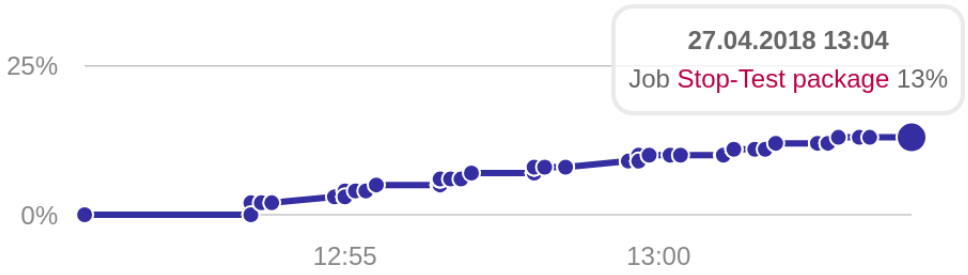
Obrázek 7.10: Velikost dílčích úloh při odpojování klientů, klienti různými barvami



Obrázek 7.11: Průběh útoku v čase při odpojování klientů



Obrázek 7.12: Velikost dílčích úloh při testu pozastavení výpočtu, klienti různými barvami



Obrázek 7.13: Průběh útoku v čase při pozastavení výpočtu

Kapitola 8

Závěr

V této práci byl proveden rozbor aktuálního stavu nástroje Fitcrack a systému BOINC. Ten je využíván pro distribuci úloh obnovy hesel na více výpočetních uzlů skrz síť LAN či Internet. Dále byla provedena analýza nástroje hashcat spolu se všemi jeho základními typy útoků. Na základě těchto znalostí jsme navrhli úpravy serverové části nástroje Fitcrack, které zajistí kompatibilitu s nástrojem hashcat na klientské části. Tyto úpravy zahrnovaly návrh komunikačního protokolu mezi serverem a klienty, strategie generování všech popsaných útoků v distribuovaném prostředí, odpovídající návrh schématu MySQL databáze a dodání dalších funkcionalit do nástroje Fitcrack. Mezi ty řadíme například automatickou detekci formátu obnovovaného souboru nástrojem XtoHashcat, znovupřidělení nedokončených úloh, korektní zastavení distribuovaného výpočtu nebo automatické měření výkonu u všech podporovaných formátů nově připojených klientů.

Takto navržené řešení nám umožní využít sílu všech útoků nástroje hashcat v distribuovaném prostředí. Díky použitému systému BOINC se jedná o prostředí robustní a spolehlivé. S pomocí zmíněných navržených funkcionalit může navíc tento systém poskytovat uživatelsky přívětivé rozhraní, které bude nabízet odhadovaný čas vytvořeného distribuovaného útoku ještě před jeho spuštěním, a které bude velké množství akcí automatizovat a zcela odstíní uživatele od argumentů příkazové řádky nástroje hashcat. Návrh modulu Generator, jako hlavního koordinátora distribuce dílčích úloh, je modulární a v budoucnu tak bude možné vytvořit komplexnější přístupy k plánování útoků.

Implementace modulů Generator a Assimilator byla provedena v jazyce C++ a to s využitím modulárního objektového návrhu a moderních prostředků, jako jsou chytré ukazatele nebo šablony. Díky tomu je kód lépe čitelný a v budoucnu jednoduše rozšiřitelný. Při implementaci nástroje XtoHashcat v jazyce Python 3 byl rovněž kladen důraz na možnost jednoduchého dodání dalších podporovaných formátů.

V provedených experimentech jsme prokázali přínos této práce. Nástroj Fitcrack dokáže s využitím nástroje hashcat provádět obnovu hesel nesrovnatelně rychleji, je použitelný jak na operačním systému Windows, tak Linux a podporuje stovky nových formátů. Dále jsme ukázali, že nástroj Fitcrack dokáže plánovat a distribuovat všechny základní útoky nástroje hashcat. V závislosti na zvoleném formátu a délce dílčí úlohy se může oproti sekvenčnímu zpracování jednat až o lineární zrychlení s rostoucím počtem připojovaných klientů.

Budoucí vývoj

I přes úspěšnou integraci nástroje hashcat do serverové části zbývá dokončit systém Fitcrack jako celek. Především tedy klientský nástroj Runner pro řízení nástroje hashcat a webové rozhraní pro ovládání výpočtu. S tím souvisí také vytvoření instalačního skriptu, který by automatizoval nasazení systému Fitcrack na vybraný výpočetní cluster. Aktuálně je tento postup značně zdlouhavý, složitý a zkušenému uživateli zabere několik hodin.

Dále by bylo vhodné detailně prostudovat provedené změny v nové verzi nástroje hashcat 4.1.0 a jejich možný vliv při využití v systému Fitcrack. Kromě rozšíření o nové formáty totiž došlo například k dodání podpory lámání až 256-bajtových hesel a hešů nebo ke změně formátu často zmiňovaného *hcstat* souboru. Ten je nově mnohonásobně menší, což je výhodné při jeho distribuci na klientské stanice. Na druhou stranu však není kompatibilní se staršími verzemi nástroje hashcat.

Systém Fitcrack aktuálně podporuje lámání balíčku, který představuje jediné heslo, respektive heš. V reálném světě ale může být žádoucí zkoušet velké množství těchto hešů a to v rámci jednoho útoku. Pokud totiž složitým procesem získáme odpovídající heš k heslu ze slovníku, je rychlejší porovnat ho s více kandidáty, než pro každého kandidáta tento proces výpočtu heše opakovat. Dalším krokem ve vývoji nástroje Fitcrack tedy bude podpora více vstupních hešů v jednom útoku.

Při dalším vývoji by se dobře uplatnil BOINC démon Trickler, který by periodicky komunikoval se všemi klienty. Mohl by tak například zjišťovat stav výpočtu v reálném čase, bez nutnosti čekat na dokončení dílčí úlohy. Navíc bychom takto mohli detekovat, že se klient odpojil nebo jiným způsobem ukončil výpočet.

Kromě přidávání nových funkcionalit by bylo žádoucí provést optimalizace těch současných. Jedná se například o minimalizaci režie komunikace mezi serverem a klienty a to správnou konfigurací serverových skriptů a klientských stanic. Dále může optimalizace probíhat na úrovni lámání jednotlivých formátů.

Literatura

[1] Anderson, D. P.: Boinc: A system for public-resource computing and storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, IEEE, 2004, ISBN 0-7695-2256-4, s. 4–10.

[2] Anderson, D. P.; Cobb, J.; Korpela, E.; aj.: SETI@home: an experiment in public-resource computing. *Communications of the ACM*, ročník 45, č. 11, 2002: s. 56–61, ISSN 0001-0782.

[3] Anderson, D. P.; Korpela, E.; Walton, R.: High-performance task distribution for volunteer computing. In *First International Conference on e-Science and Grid Computing*, IEEE, 2005, ISBN 0-7695-2448-6, s. 196–203.

[4] Bonneau, J.; Herley, C.; Van Oorschot, P. C.; aj.: The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy*, IEEE, 2012, ISBN 978-1-4673-1244-8, s. 553–567.

[5] Gazdík, P.: *Využití heuristik při obnově hesel pomocí GPU*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=18210

[6] Herley, C.; Van Oorschot, P.: A research agenda acknowledging the persistence of passwords. *IEEE Security & Privacy*, ročník 10, č. 1, 2012: s. 28–36.

[7] Hitaj, B.; Gasti, P.; Ateniese, G.; aj.: PassGAN: A Deep Learning Approach for Password Guessing. *arXiv:1709.00440*, 2017.
URL https://arxiv.org/pdf/1709.00440.pdf

[8] Houshmand, S.; Aggarwal, S.; Flood, R.: Next gen PCFG password cracking. *IEEE Transactions on Information Forensics and Security*, ročník 10, č. 8, 2015: s. 1776–1791, ISSN 1556-6013.

[9] Hranický, R.; Holkovič, M.; Matoušek, P.; aj.: On Efficiency of Distributed Password Recovery. *The Journal of Digital Forensics, Security and Law*, ročník 11, č. 2, 2016: s. 79–96, ISSN 1558-7215.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11276

[10] Hranický, R.; Matoušek, P.; Ryšavý, O.; aj.: Experimental Evaluation of Password Recovery in Encrypted Documents. In *Proceedings of ICISSP 2016*, SciTePress - Science and Technology Publications, 2016, ISBN 978-989-758-167-0, s. 299–306.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11052

[11] Hranický, R.; Zobal, L.; Večeřa, V.: Distribuovaná obnova hesel. Technická zpráva, FIT-TR-2017-04, 2017.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11568

[12] Hranický, R.; Zobal, L.; Večeřa, V.; aj.: Distributed Password Cracking in a Hybrid Environment. In *Proceedings of SPI 2017*, Brno University of Defence, 2017, ISBN 978-80-7231-414-0, s. 75–90.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=11358

[13] Jiránek, K.: *Generování hesel na základě pravidel*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2017.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=19900

[14] Morris, R.; Thompson, K.: Password security: A case history. *Communications of the ACM*, ročník 22, č. 11, 1979: s. 594–597, ISSN 0001-0782.

[15] Narayanan, A.; Shmatikov, V.: Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM conference on Computer and communications security*, ACM, 2005, ISBN 1-59593-226-7, s. 364–372.

[16] Provos, N.; Mazieres, D.: A Future-Adaptable Password Scheme. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 1999, s. 81–91.

[17] Schmied, J.: *GPU akcelerované prolamování šifer*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2014.
URL http://www.fit.vutbr.cz/study/DP/DP.php?id=16858

[18] Weir, M.; Aggarwal, S.; De Medeiros, B.; aj.: Password cracking using probabilistic context-free grammars. In *30th IEEE Symposium on Security and Privacy*, IEEE, 2009, ISBN 978-0-7695-3633-0, s. 391–405.

[19] Zobal, L.: *Obnova hesel dokumentů Microsoft Office s využitím GPU*. Bakalářská práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.
URL http://www.fit.vutbr.cz/study/DP/BP.php?id=18341

Příloha A

Obsah CD

Na přiloženém CD se nachází:

- tato práce v digitální podobě,
- zdrojové kódy této práce v LaTeXu,
- zdrojové kódy programů Assimilator a Generator, spolu se zdrojovými kódy systému BOINC pro kompilaci,
- binární soubory programů Assimilator a Generator ve spustitelné podobě,
- skripty SQL pro vytvoření databáze a import potřebných spouští,
- zdrojové kódy programu XtoHashcat,
- tabulky se získanými daty z experimentů,
- soubor README s instrukcemi k vytvoření Fitcrack projektu.

Příloha B

Struktura databáze MySQL

B.1 fc_benchmark

Jedná se o nově doplněnou tabulku, která slouží pro ukládání výsledků úloh typu benchmark a kompletního benchmarku. Tabulka obsahuje kombinaci typ heše a jeho výkon pro každého naměřeného klienta. Díky těmto informacím pak můžeme provádět výpočty odhadovaného času běhu vybraného útoku. Sloupce tabulky mají následující význam:

- **id** – primární klíč tabulky,
- **boinc_host_id** – reálné ID klienta uvedené v BOINC tabulce host,
- **hash_type** – typ heše v hashcat formátu `--attack-mode`,
- **power** – poslední naměřený výkon klienta vzhledem k danému formátu, v počtu reálných hesel za sekundu,
- **last_update** – datum a čas poslední aktualizace záznamu.

B.2 fc_hashcache

Jedná se opět o zcela novou tabulku, do které jsou ukládána nalezená hesla v kombinaci s původním vstupem. To pak umožňuje uživateli prohledat všechna dříve nalezené hesla či heše před tím, než začne proces samotné obnovy. Schéma této tabulky je následující:

- **id** – primární klíč tabulky,
- **hash_type** – typ heše v hashcat formátu `--attack-mode`,
- **hash** – původní vstup do nástroje hashcat,
- **result** – získané heslo či vstup heše,
- **added** – datum a čas přidání položky.

B.3 fc_host

Tato tabulka obsahuje seznam aktivních klientů, kteří se aktuálně podílejí na výpočtu. Pomocí této tabulky jsou jednotliví klienti svázáni s balíčky a je zde uveden také jejich současný výkon. Schéma této tabulky je následující:

- **id** – primární klíč tabulky,
- **boinc_host_id** – reálné ID klienta uvedené v BOINC tabulce host,
- **power** – poslední naměřený výkon klienta vzhledem k danému formátu, v počtu reálných hesel za sekundu,
- **package_id** – cizí klíč do tabulky fc_package,
- **status** – stav klienta, může nabývat těchto hodnot:
  - **0** – klient provádí benchmarku nebo čeká na jeho přidělení,
  - **1** – klient provádí klasický výpočet,
  - **3** – klient dokončil všechnu přidělenou práci a již mu není generována další,
  - **4** – na klientu došlo k chybě,
- **time** – datum a čas přidání položky.

B.4 fc_host_activity

Tato tabulka slouží k provázání klientů a balíčků. Určuje tedy, kteří klienti se mají účastnit kterého výpočtu. Schéma této tabulky je následující:

- **id** – primární klíč tabulky,
- **boinc_host_id** – reálné ID klienta uvedené v BOINC tabulce host,
- **package_id** – cizí klíč do tabulky fc_package.

B.5 fc_job

Tato tabulka obsahuje záznamy o všech provedených úlohách. Nalezneme zde všechny potřebné informace o tom, kdo úlohu prováděl, k jakému balíčku se váže, jak je veliká, zda a kdy byla dokončena apod. Schéma této tabulky je následující:

- **id** – primární klíč tabulky,
- **package_id** – cizí klíč do tabulky fc_package,
- **workunit_id** – reálné ID úlohy uvedené v BOINC tabulce workunit,
- **host_id** – ID klienta z tabulky fc_host,
- **boinc_host_id** – reálné ID klienta uvedené v BOINC tabulce host,
- **start_index** – index stavového prostoru, kde úloha začíná,
- **start_index_2** – index pro první slovník v kombinačním útoku; použitý jen pokud jsou klienti příliš slabí nebo slovníky příliš velké,
- **hc_keyspace** – stavový prostor hesel úlohy v hashcat počítání; pro slovníkový a kombinační útok se jedná o počet reálných hesel,
- **passwords_verified** – průběh úlohy – pro maskový útok udává počet již ověřených hashcat indexů, pro slovníkový a kombinační útok se jedná o počet reálných hesel,
- **mask_id** – odkaz na masku do tabulky fc_mask, se kterou daná úloha počítá,

- **duplicated** – příznak, zda byla úloha kvůli jejímu nedokončení rozdělena na více úloh,
- **duplicate** – pokud je duplicated 1, obsahuje ID původní úlohy,
- **time** – datum a čas přidání položky,
- **cracking_time** – čas běhu dané úlohy,
- **retry** – příznak, zda se jedná o znovu přiřazenou úlohu,
- **finished** – příznak, zda byla úloha dokončena.

B.6 fc_mask

Tato tabulka obsahuje seznam mask, které byly kdy použity ve slovníkových útocích. Také zde nalezneme vazbu mezi každou maskou a balíčkem. Schéma této tabulky je následující:

- **id** – primární klíč tabulky,
- **package_id** – cizí klíč do tabulky fc_package,
- **mask** – řetězec s hashcat maskou,
- **current_index** – aktuální index průchodu stavovým prostorem masky,
- **keyspace** – reálný počet hesel v masce,
- **hc_keyspace** – počet hashcat indexů v masce.

B.7 fc_package

Tato tabulka obsahuje seznam balíčků, tedy souborů určených k obnově. Nalezneme zde všechny potřebné informace nutné pro provedení distribuovaného útoku. Schéma této tabulky je následující:

- **id** – primární klíč tabulky,
- **token** – unikátní identifikátor sezení ,
- **attack_mode** – typ útoku hashcatu:
  - **0** – slovníkový útok s případnými pravidly,
  - **1** – kombinační útok, případně hybridní útok, převedený na kombinační,
  - **3** – maskový útok s případným *.hcstat* souborem,
- **attack_submode** – typ podútoku, v kombinaci s předchozí hodnotou má následující význam:
  - **0 + 0** – slovníkový útok,
  - **0 + 1** – slovníkový útok s pravidly,
  - **1 + 0** – kombinační útok,
  - **1 + 1** – kombinační útok s levým pravidlem,
  - **1 + 2** – kombinační útok s pravým pravidlem,
  - **1 + 3** – kombinační útok s oběma pravidly,
  - **3 + 0** – maskový útok,

  - **3 + 1** – maskový útok s *.hcstat* souborem, využívající 2D matici,
  - **3 + 2** – maskový útok s *.hcstat* souborem, využívající 3D matici,

- **hash_type** – typ heše v hashcat formátu,
- **hash** – vstup do nástroje hashcat,
- **status** – stav balíčku:
  - **0** – neprobíhá výpočet,
  - **1** – obnova byla dokončeno a heslo nalezeno,
  - **2** – byl prohledán celý zadaný stavový prostor hesel, ale správné heslo zde nebylo,
  - **3** – balíček obsahuje chybný vstup,
  - **4** – balíček byl ukončen z důvodu vypršení nastaveného času,
  - **10** – probíhá výpočet,
  - **12** – výpočet dobíhá, někteří klienti počítají, ale již se negenerují nové úlohy,
- **result** – správné heslo, pokud bylo nalezeno,
- **keyspace** – počet všech možných hesel,
- **hc_keyspace** – stavový prostor v hashcat indexech,
- **passwords_verified** – počet již ověřených hashcat indexů, pro slovníkový a kombinační útok počet ověřených reálných hesel,
- **current_index** – aktuální index stavového prostoru hashcatu, pro kombinační útok se jedná o index v 2. slovníku,
- **current_index_2** – index pro pohyb v prvním slovníku v kombinačním útoku,
- **time** – datum a čas přidání položky,
- **name** – název balíčku,
- **comment** – komentář k balíčku,
- **time_start** – čas zahájení obnovy,
- **time_end** – čas ukončení obnovy,
- **cracking_time** – celkový čas, který v součtu strávili klienti výpočtem,
- **seconds_per_job** – doba v sekundách, po kterou má běžet jedna úloha,
- **config** – TLV formát pro předávání potřebných hodnot klientům,
- **dict1** – název souboru se slovníkem pro slovníkový a kombinační útok,
- **dict2** – název souboru s druhým slovníkem pro kombinační útok,
- **charset1** – uživatelem definovaná znaková sada #1 v kódování base64,
- **charset2** – uživatelem definovaná znaková sada #2 v kódování base64,
- **charset3** – uživatelem definovaná znaková sada #3 v kódování base64,
- **charset4** – uživatelem definovaná znaková sada #4 v kódování base64,
- **rules** – název souboru s pravidly,
- **rule_left** – pravidlo pro modifikaci levého slovníku,
- **rule_right** – pravidlo pro modifikaci pravého slovníku,

- **markov_hcstat** – název souboru s *.hcstat* souborem,
- **markov_threshold** – práh pro Markovovu matici,
- **replicate_factor** – replikační faktor pro daný balíček.

B.8 fc_settings

Tato tabulka slouží pro nastavení hodnot ovlivňujících výpočet napříč různými balíčky, případně některých standardních hodnot. Schéma této tabulky je následující:

- **id** – primární klíč tabulky,
- **delete_finished_jobs** – příznak, zda se mají dokončené úlohy mazat z tabulky fc_job,
- **default_seconds_per_job** – výchozí doba trvání jedné úlohy,
- **default_replicate_factor** – výchozí replikační faktor úloh,
- **default_verify_hash_format** – příznak, zda se má při vytváření balíčku ověřit vstupní heš,
- **default_check_hashcache** – příznak, zda se má při vytváření balíčku ověřit hashcache, viz výše,
- **default_job_timeout_factor** – výchozí doba pro prohlášení úlohy za nedokončenou,
- **default_bench_all** – příznak, zda se má při připojení nového klient provést automaticky kompletní benchmark.

Příloha C

Popis možných výsledků výpočtu

C.1 Benchmark

Jedná se o úlohu, která měří výkon daného klienta vzhledem k danému formátu. Identifikující znak na prvním řádku je *b*. Existují pouze dva možné výsledky této úlohy. Pokud se benchmark povede, zašle se na druhém řádku stavový kód 0. Kompletní zpráva má následující tvar:

- b
- 0
- <Změřený výkon klienta v heších za sekundu> – datový typ integer
- <Doba benchmarku v sekundách> – datový typ double

Při selhání benchmarku je zaslán stavový kód 4. Celá zpráva pak vypadá následovně:

- b
- 4
- <Návratový kód nástroje hashcat> – datový typ integer
- <Chybová hláška nástroje hashcat> – datový typ string

C.2 Výpočet

Jedná se o výpočetní úlohu, která má za úkol najít heslo v přiděleném stavovém prostoru s využitím daného útoku. Pokud nenastane neočekávaná chyba, heslo je ve přiděleném prostoru buď nalezeno nebo nikoliv. Pokud heslo nalezeno je, zasílá se stavový kód 0 a formát zprávy pak vypadá následovně:

- n
- 0
- <Získané heslo/heš v kódování base64> – datový typ string
- <Doba výpočtu v sekundách> – datový typ double

Pokud naopak heslo nalezeno není, je zaslán stavový kód 1. Formát zprávy je stejný, pouze je vynechána informace o získaném hesle:

- n
- 1
- <Doba výpočtu v sekundách> – datový typ double

Při chybě rozlišujeme dva typy. První je způsobena chybným stavem balíčku na serveru. Jsou například zaslány chybné argumenty nebo nějaké chybí. Na takovou zprávu klient odpoví stavovým kódem 3 společně s informacemi o chybovém výstupu nástroje hashcat:

- n
- 3
- <Návratový kód nástroje hashcat> – datový typ integer
- <Chybová hláška nástroje hashcat> – datový typ string

Pokud je chyba na straně klienta, například ve formě chybějících ovladačů, pádu nástroje hashcat apod., je zaslán stavový kód 4:

- n
- 4
- <Návratový kód nástroje hashcat> – datový typ integer
- <Chybová hláška nástroje hashcat> – datový typ string

C.3 Kompletní benchmark

Jedná se o úlohu, která má za úkol provést benchmark všech formátů a následně zaslat výsledky serveru. Zde jsou možné situace podobné klasickému benchmarku. Pokud se povede měření alespoň nějakých formátů, je odeslán výsledek se stavovým kódem 0. U formátů, jejichž měření selhalo, je uvedena rychlost obnovy rovná nule. Výsledek vypadá následovně:

- a
- 0
- <Doba výpočtu v sekundách> – datový typ double
- <Formát nástroje hashcat>:<Výkon klienta> – datové typy integer
- <Formát nástroje hashcat>:<Výkon klienta> – datové typy integer
- ... (*Následuje seznam všech podporovaných formátů*)

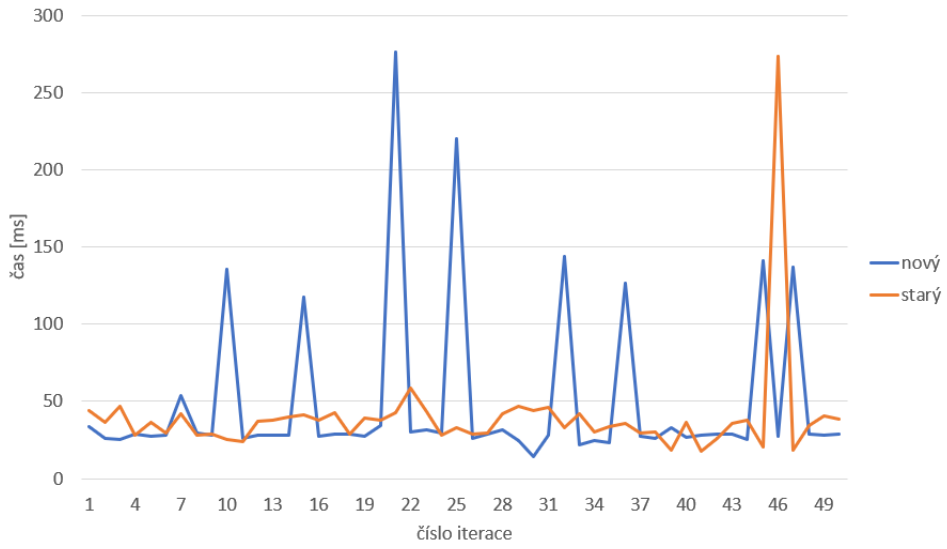
Pokud tato úloha z nějakého důvodu selže, například kvůli pádu nástroje hashcat, je zaslána zpráva v následujícím formátu:

- a
- 4
- <Informace o tom, co se pokazilo> – datový typ string

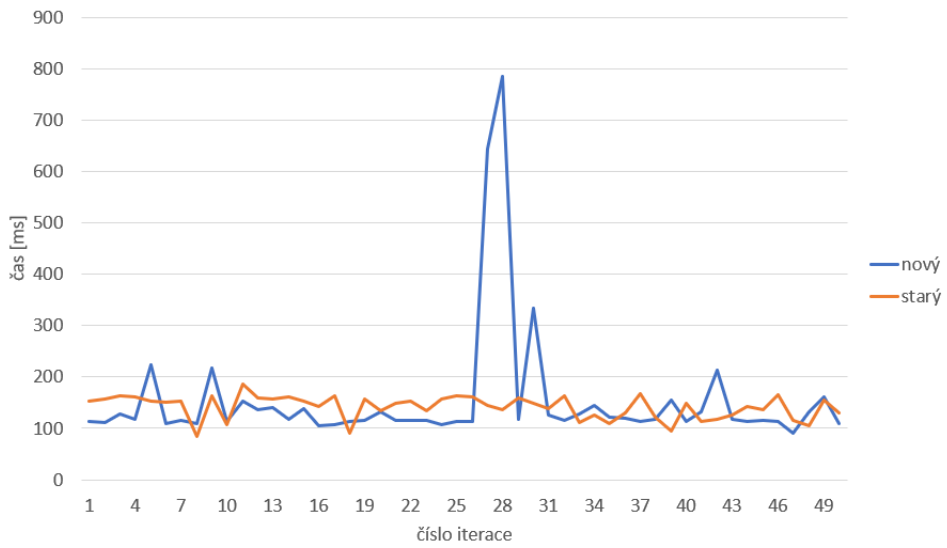
Příloha D

Porovnání starého a nového programu Generator

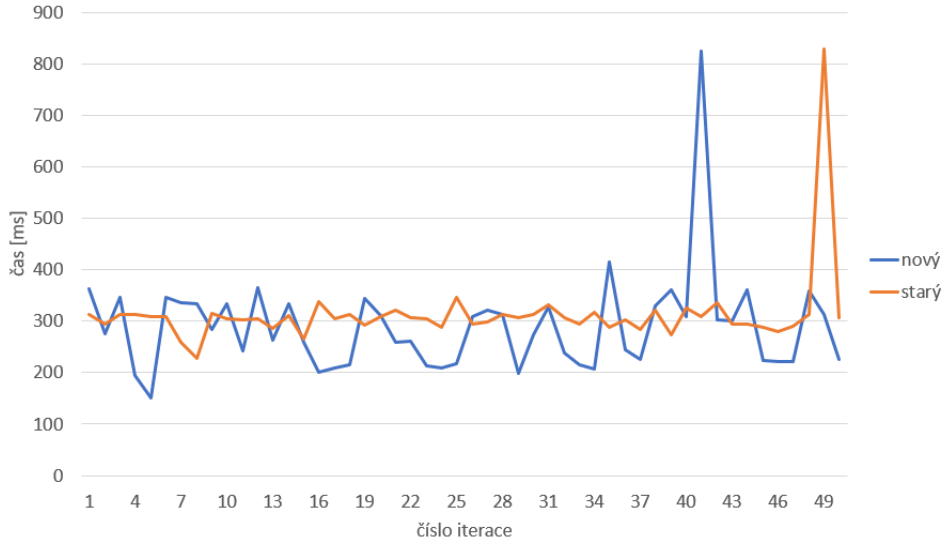
V následující příloze se nachází grafy, zobrazující všechna získaná data z měření popsaného sekcí 7.2. Na každém grafu vidíme doby jednotlivých iterací pro různé počty klientů jak pro nový, tak starý program Generator.



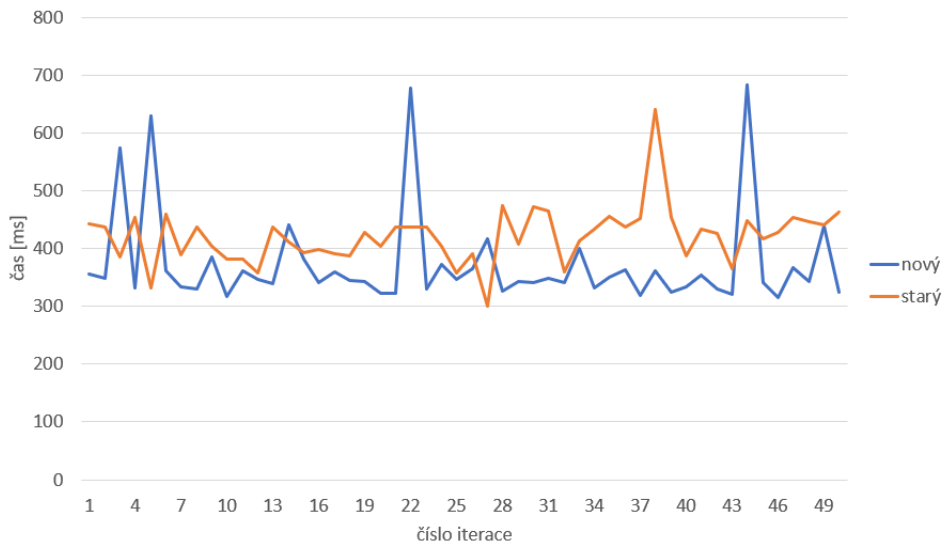
Obrázek D.1: Měření doby iterace pro 1 připojeného klienta



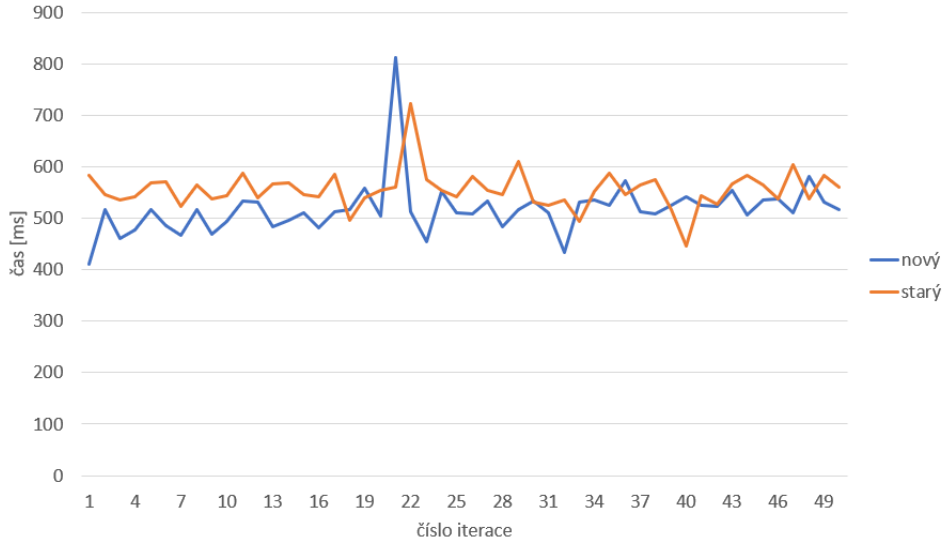
Obrázek D.2: Měření doby iterace pro 5 připojených klientů



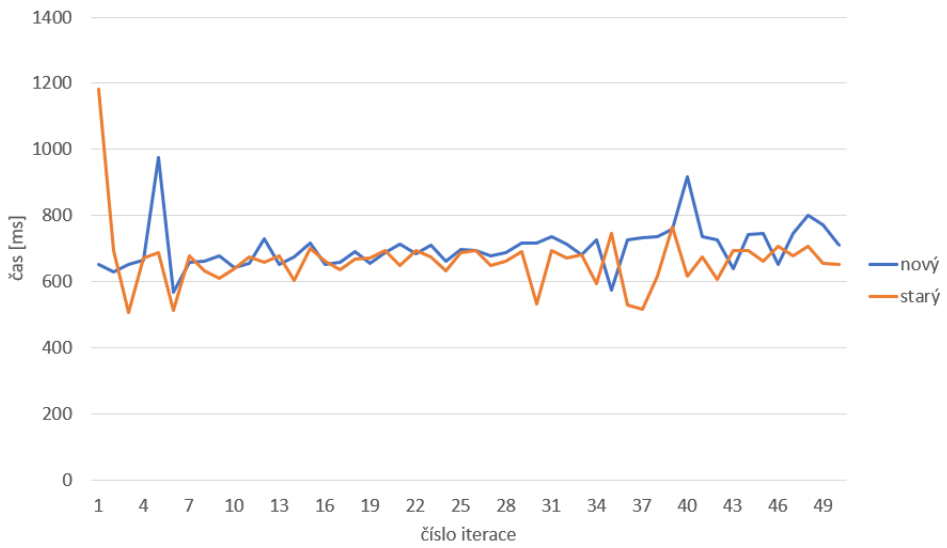
Obrázek D.3: Měření doby iterace pro 10 připojených klientů



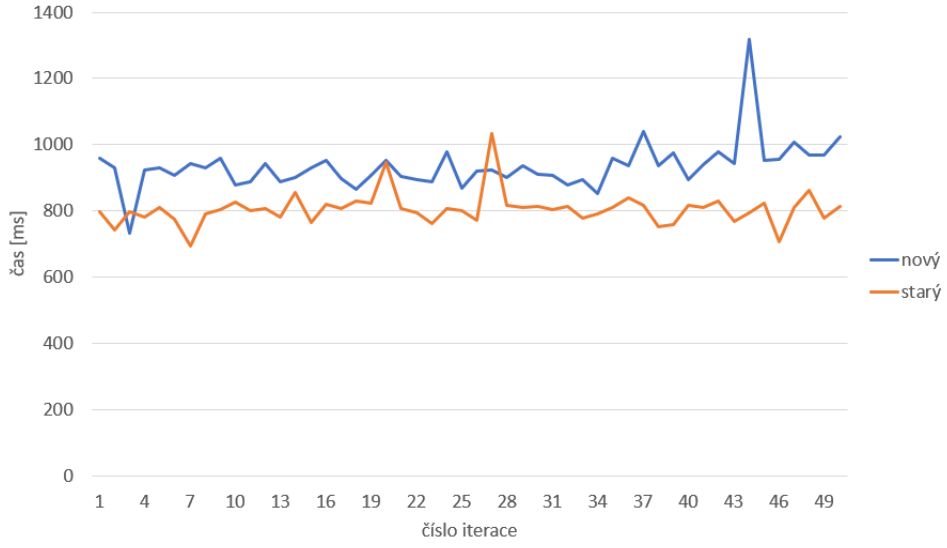
Obrázek D.4: Měření doby iterace pro 15 připojených klientů



Obrázek D.5: Měření doby iterace pro 20 připojených klientů



Obrázek D.6: Měření doby iterace pro 25 připojených klientů



Obrázek D.7: Měření doby iterace pro 30 připojených klientů